CS251
Data structures. Summer 2016

- Gustavo Rodriguez-Rivera
  grr@purdue.edu
  LWSN 1203

- www.cs.purdue.edu/homes/cs251

- Grades
  - 25% - Midterm
  - 25% - Final
  - 10% - Attendance.
  - 40% - Labs + HW's

- Post questions in Piazza

Goal

- In this course you will learn how to represent different types of data in optimal way in computer data structures.
- It is also an introduction to computer algorithms.

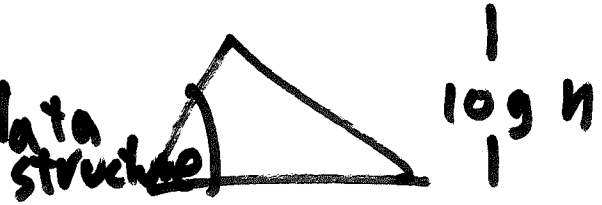
Algorithm ② + Data Structure ① = program

* N. Wirth

- You will also improve your programming skills.

Content

- Analysis Tools
- Stacks, Queues
- Vectors, Lists, Sequences.
- Trees
- Priority Queues (Heap data structure)



- Dictionaries
- Search Trees (AVL trees)
- Sorting (QuickSort, Merge Sort, Heap Sort, 2/3 trees, RB trees, Bucket Sort, Radix Sort).
- Text processing / Strings
- Graphs / Graph Algorithms.

Math Review

$$a^{b+c} = a^b a^c$$

$$a^{bc} = (a^b)^c$$

$$\frac{a^b}{a^c} = a^{b-c}$$

$$\log_a b = c \iff a^c = b$$

$$\log_b xy = \log_b x + \log_b y$$

$$\log_b \frac{x}{y} = \log_b x - \log_b y$$

$$\log_b x^\alpha = \alpha \log_b x$$

Floor $\lfloor x \rfloor$ = the largest integer $\leq x$

Ceiling $\lceil x \rceil$ = the smallest integer $\geq x$

Summations

$$\sum_{i=s}^{t} f(i) = f(s) + f(s+1) + f(s+2) \ldots f(t)$$

s → start index
t → end index.

Geometric Progression

$$\sum_{i=0}^{n} a^i = 1 + a^1 + a^2 + \ldots a^n = s(a,n)$$

a^0

$$as(a,n) = a + a^2 + a^3 + \ldots a^{n+1}$$

$$as(a,n) - s(a,n) = \cancel{a} + \cancel{a^2} + \cancel{a^3} + \ldots \cancel{+} a^{n+1}$$
$$-1 - \cancel{a} - \cancel{a^2} - \cancel{a^3} \ldots \cancel{a^n}$$

$$as(a,n) - s(a,n) = a^{n+1} - 1$$

$$s(a,n)(a-1) = a^{n+1} - 1$$

$$\boxed{s(a,n) = \frac{a^{n+1} - 1}{a-1}}$$

Arithmetic Progressions

$$A(i) = \sum_{i=1}^{n} i = 1+2+3+\dots n = \frac{n^2+n}{2}$$

$$= \frac{n(n+1)}{2}$$

When measuring an algorithm we measure two parameters

- Time
- Space

* Time (secs) - How long does the algorithm take.
  It depends On:
  + Algorithm itself
  + Hardware (machine).
  + Input size and type of input.
  + Programmer / Language / Compiler optimizer
* Space (bytes) - Computer storage used by the data structure

In this course we put more emphasis in Time rather than Space.

* space is important but not as important as time.

Types of running time.

- Worst Case - Time that algorithm takes with most difficult input.
- Average Case - Average time across all inputs.
- Best Case - Time that algorithm takes with easiest input.

- Time is measured experimentally using inputs of different sizes and types

- You may use the time command in Unix
  time command (prog) → prints how long command take.

Eg.

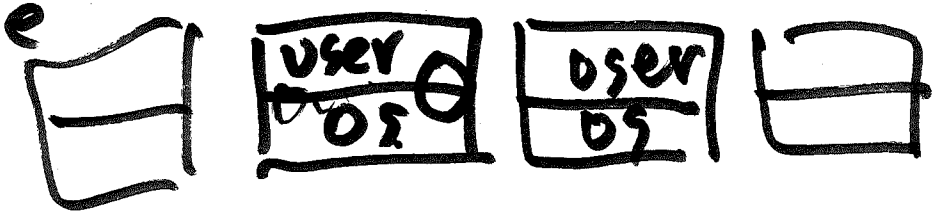
time gcc myprog.c → How long compilation takes.

User space time ← user : 2 secs

OS time (time in kernel) ← sys : .5 secs

← total : 3 secs

Wall clock time



single CPU

user + sys ≤ total

N CPU computer

user + sys ≤ N total

- Experimental studies of how long a program or algorithm takes have limitations.
  - It is necessary to implement the algorithm.
  - There is a limited set of inputs used. It is impossible to measure execution for -all- inputs
  - To compare algorithms they need to run in the same machine, same compiler, same input etc. Same language etc.

- Pseudo Code
  - High-level description of an algorithm.
  - Enough information to come up with a program that implements the algorithm.

We will develop a technique to compare algorithms that does not require an experimental analysis. This technique is called "Asymptotic Analysis".

Asymptotic Analysis

We will use:

- The worst case
- Very large input size $n \to \infty$

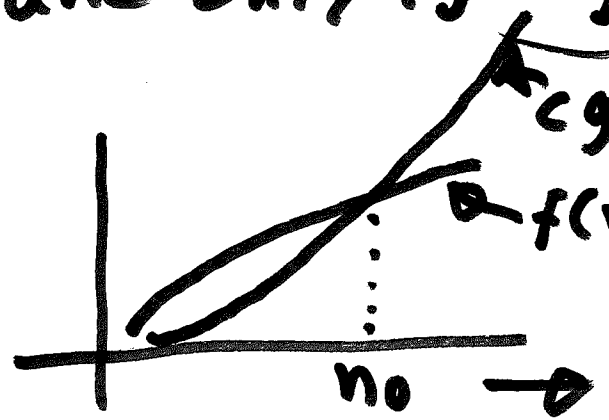
Steps:

1. We count the operations for an input size n.
2. We assume that n is very large ($n \to \infty$) and eliminate the components that become unimportant as n grows to infinite.

Asymptotic notation

Also called "Big-Oh" notation.

Given functions $f(n)$ and $g(n)$
we say that $f(n)$ is $O(g(n))$
if and only if $f(n) \leq c\,g(n)$ for $n > n_0$
and for some c.



$f(n) \leq c\,g(n)$ for $n > n_0$

then $f(n)$ is $O(g(n))$

Example:

Execution time $f(n) = .000001\,n^2 + 100n + 1000$

Execution time $f(n) = O(?)$

Execution Time $(n=1) = .000001(1)^2 + 100(1) + 1000$

$.000001 + 100 + 1000 \sim 1100$

← most significant

Execution Time $(n=1000) = .000001(1000)^2 + 100(1000) + 1000$

10^{-6} 10^{3}

↓

$1 + 100{,}000 + 1000$

← most significant

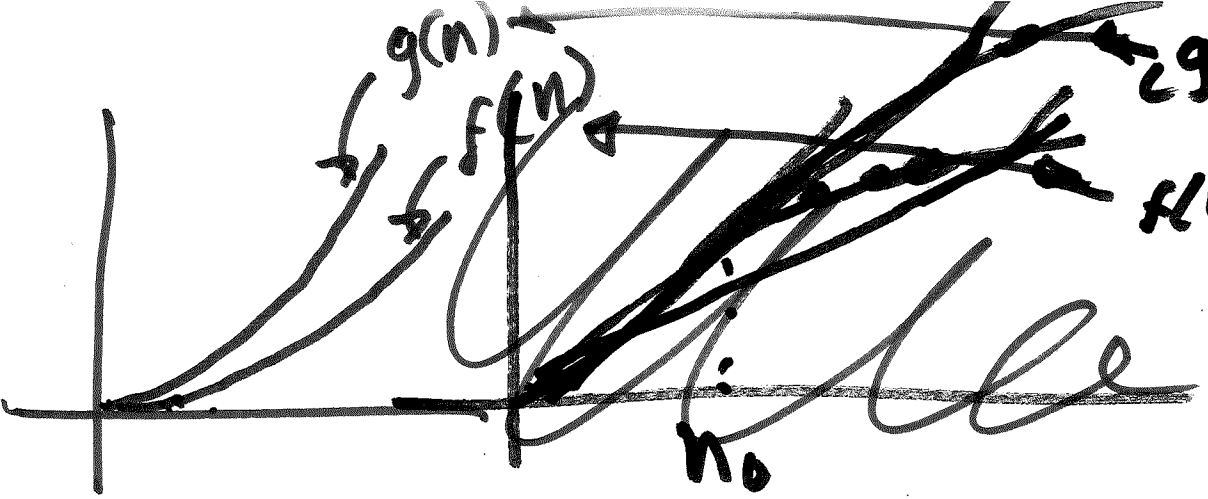
Execution time $(n=10^{20}) = .000001(10^{20})^2 + 100(10^{20}) + 1000$

10^{-6} $10^{40} + 10^{22} + 100c$

$10^{34} + 10^{22} + 1000$

← most important.

Execution Time $= O(n^2)$



$cg(n) = cn^2$

$f(n) = .000001n^2 + 100n + 10000$

$f(n) = O(n^2)$

$f(n)$ is ~~also~~ technically also $O(n^3)$ or $O(n^4)$ but we will use the $g(n)$ that most accurately describe how $f(n)$ grows.

Goal of Asymptotic analysis is to get rid of unneeded information.

- It is expected that the approximation should be as small in order as possible.

$f(n) = 7n - 3$ is also $O(n^5)$ but we choose instead $O(n)$.

- Drop lower order terms and constant factors

$7n-3$ is $O(n)$

$8n^2\log n + 5n^2$ is $O(n^2 \log n)$

$7\log n + n$ is $O(n)$

Special cases of algorithms

$O(\log n)$	logarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^k)$	polynomial
$O(a^n)$	$a>1$ exponential

$5n^{53} + (1.03)^n \rightarrow O(a^n)$ exponential

more important $n \to \infty$

Example

Sort an array of n numbers

Algorithm bubblesort(A, n)

Input: array of n integers

Output: Sorted array.

```
for(i=0; i<n-1; i++) {                    <- O(n)
    for(J=0; J<n-i-1; J++) {              <- O(n)
        if(A[J] > A[J+1]) {
            //Need to swap
            int tmp = A[J];
            A[J] = A[J+1];
            A[J+1] = tmp;
        }
    }
}
```

-> $O(n^2)$

Time = O(?)

0	1	3
1	8	8
2	3	3
3	2	7
4	7	4
5	4	8

J=0, J=4

i=0	n-1
i=1	+ n-2
	+ n-3
	⋮
i=n-1	0

time = $\sum_{i=0}^{n-1}$ ((n-1) + (n-2) + (n-3) ... 0) =

$$time = c\left[n(n-1) - \sum_{i=1}^{n-1} i\right]$$

$$= c\left[n^2 - n - \frac{(n-1)(n)}{2}\right]$$

$$= c\left[\frac{2n^2 - 2n - n^2 + n}{2}\right]$$

$$= c\left[\frac{n^2 - n}{2}\right] = O(n^2)$$

$$\sum_{i=1}^{n-1} i = \frac{(n-1)(n)}{2}$$

Stacks, Queues, Linked Lists

Stack

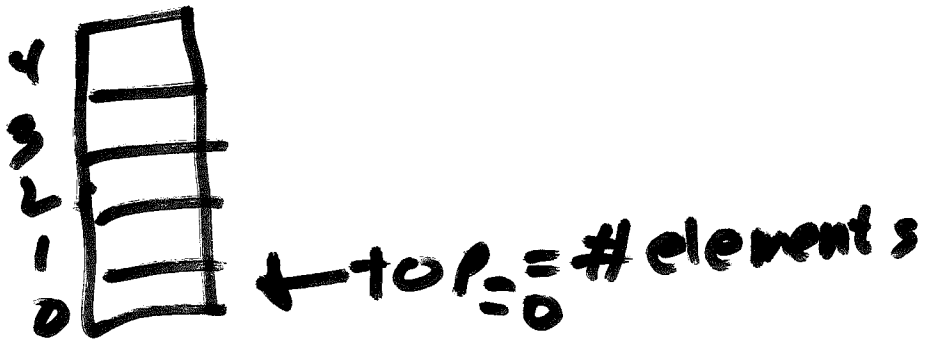
It is a container of objects where they are inserted and removed according to the last-in-first-out principle (LIFO).

Two operations:

- push(val) – inserts item at the top of stack
- val=pop() – removes item at the top of stack and returns it.

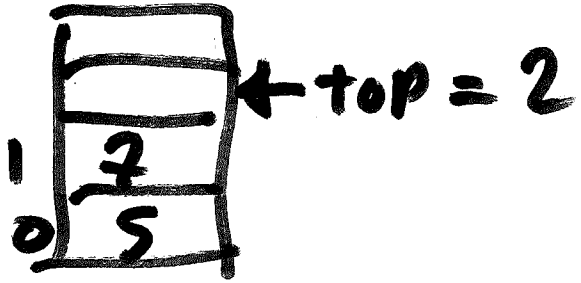
Additional methods:

- size() – # of elements in stack
- isEmpty()
- top():

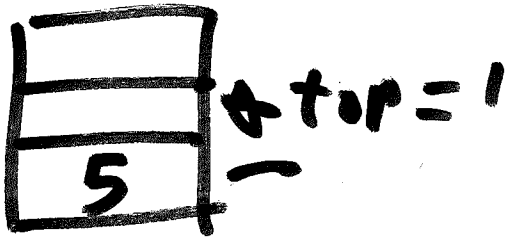


push 5

push 7

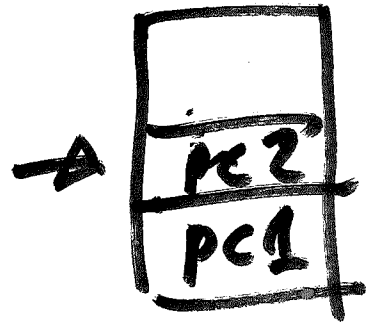


val = pop() → val == 7



Where are stacks being used?

→ Execution Stack.




```
main() {
    f1();
pc1:
}

f1() {
    f2();
pc2:
}

f2() {
    ret
}
```

Quiz 1

(20)

The function

$$f(n) = 3\log(n) + 50n + 3n^{1/2}$$

is

a) $O(n)$

b) $O(\log n)$

c) $O(n^{1/2})$

Hash Tables

A hash table is a data structure that stores pairs (key, data)

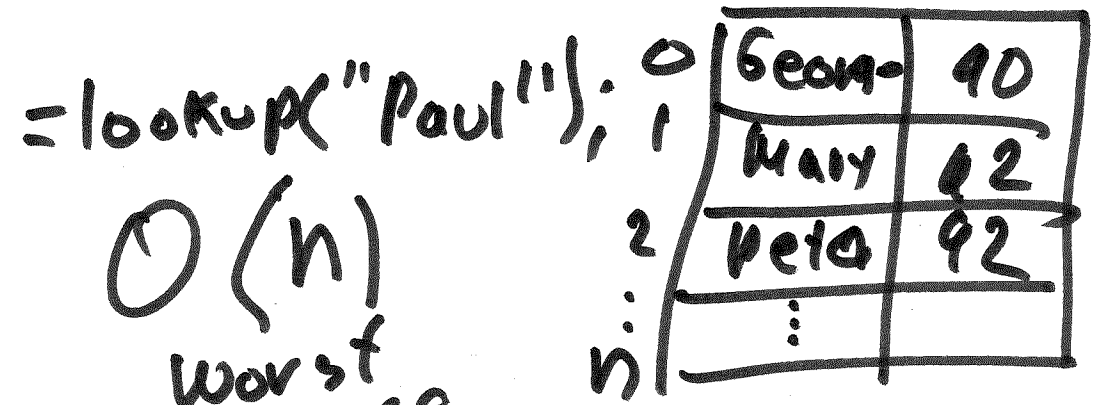
key (name)	data (grade)
Student,	grade
George	90
Mary	92
Peter	92

E.g.

key (name)	data (struct student record) address, SID, Major
George	(200 Grant, 3527, CS)
Mary	(34 Main, 2275, EE)
Peter	(43 Third, 753, Econ)

If we store table in an array, a lookup in n entries will be

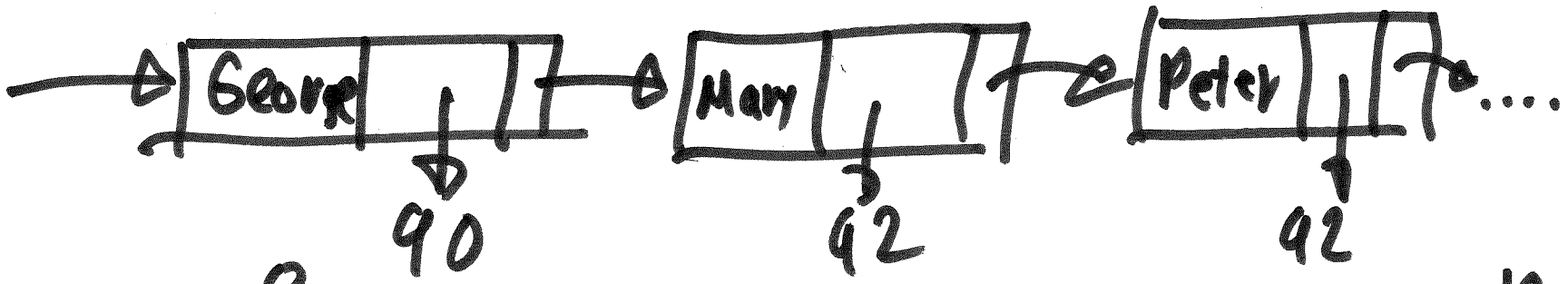
```
grade = lookup("Paul");
```



$O(n)$

worst case.

If we use linked list



also grad = lookup("Paul");

$O(n)$.

Quiz 2

Assume we allocate an array as (23)

```
array = new int [n];
```

To deallocate array we use:

a) delete array

b) free array

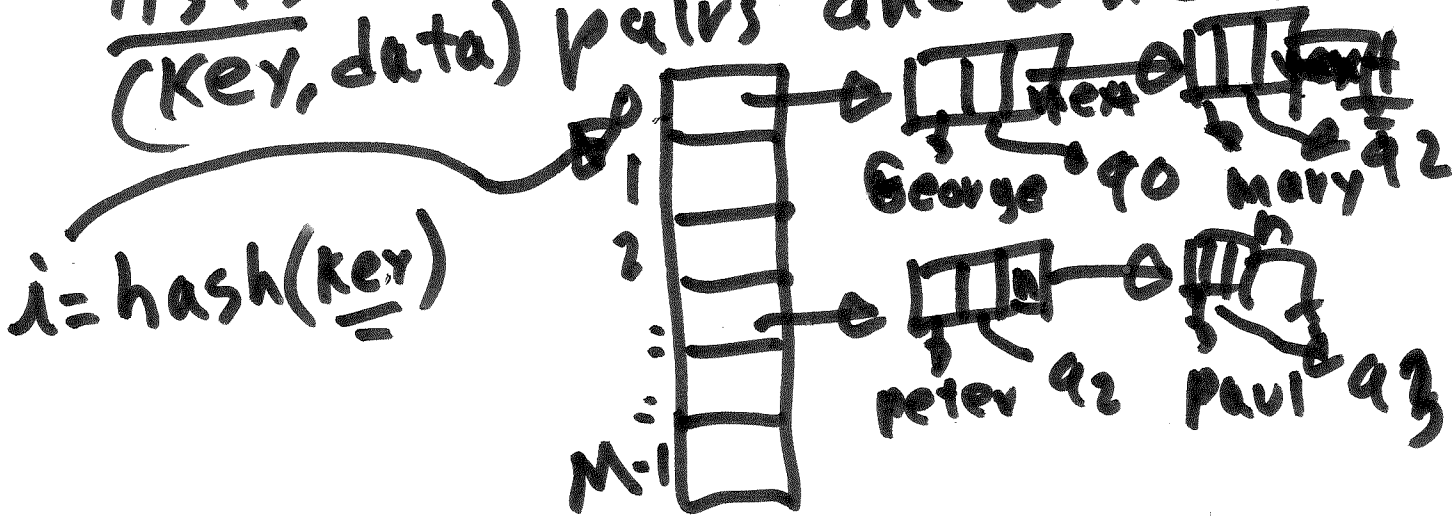
c) delete [] array

A hash table also implements a dictionary to store (key, data) pairs.

A hash table has an expectad time of O(1) for lookup operation

The worst case is O(n) but it is unlikely to happen if we follow some rules we will talk about later.

A hash table is made of an array of lists where each entry in the list have (key, data) pairs and a next.



The hash function takes a key as argument and generates an integer $0 \to M-1$.

The hash function/hash table needs to have the following properties:

1. $i = \text{hash}(\text{key})$ has to generate numbers $0 \leq i \leq M-1$
2. The hash function has to be evenly distributed, that is all numbers $0 \leq i \leq M-1$ will have the same probability from occuring for the input keys
3. The number M of buckets (lists) in hash table have to be larger or comparable to N (# of keys) $M \geq N$

If the hash table has these 3 characteristics, the expected time will be O(1).

Algorithm lookup

```
data = lookup(key):
    i = hash(key)
    list = bucket[i];
    while list != NULL && list->key != key
        list = list->next;

    if list == NULL : //reached end
        return NULL;
    else //found
        return list->data
```

Algorithm insert

```
boolean insert(key, data):
    i = hash(key)
    list = bucket[i];
    while list != NULL && list->key != key
        list = list->next;
    if list != NULL : // key exists
        // overwrite data
        list->data = data
        return true
    else : // key does not exist
        entry = new Entry
        entry->key = key
        entry->data = data
        entry->next = bucket[i];
```

```
            bucket[i] = entry;
        end
        return false
    end
```

Hash functions:

Ints
If key is an int

```
int hash(int key) {
    return key % M;
}
```

Strings
If key is a string, a hash function could be implemented by adding the ascii value of all characters in key

```
int hash(char *key){
    int sum = 0;
    for(int i = 0; key[i] != '\0'; i++){
        sum += key[i];
    }
    return sum%M;
}
```

One problem of this hash function will be that some ~~characters~~ keys with the same characters but permutated will have the same hash value. apple + pleap
In an improvement is to use the position of a character as part of the hash function.

```
int hash(char *key){
 int sum=0;
 for(int i=0;key[i]!='\0';i++){
    sum += i*key[i];
 }
 return sum % M;
}
```

Also to improve the hash table performance and have the hash function be more evenly distributed is to choose M to be prime to prevent hash values from being periodic.

```
HashTableVoid::HashTableVoid() {
    _buckets = new HashTableVoidEntry*[TableSize];
    for (int i=0; i<TableSize; i++) {
      _buckets[i] = NULL;
    }
}
```

Quiz 3

Which of the following properties are necessary to have a lookup() expected time of O(1) in a hash table:

a) hash function needs to be evenly distributed

b) $M \geq n$

c) $n \geq M$

d) a and b

e) a and c

M = ~~size of~~
M = # of buckets
n = # of keys

Another use of stacks

Matching enclosing { }, ( ), [ ]

Function isMatchOk(const char * s) {
// It returns true if S has nested
// { }, ( ), [ ], that match each other
// without interleaving

{ ( ) ( ) } [ ] ok

{ [ ] ([ ]) } ok

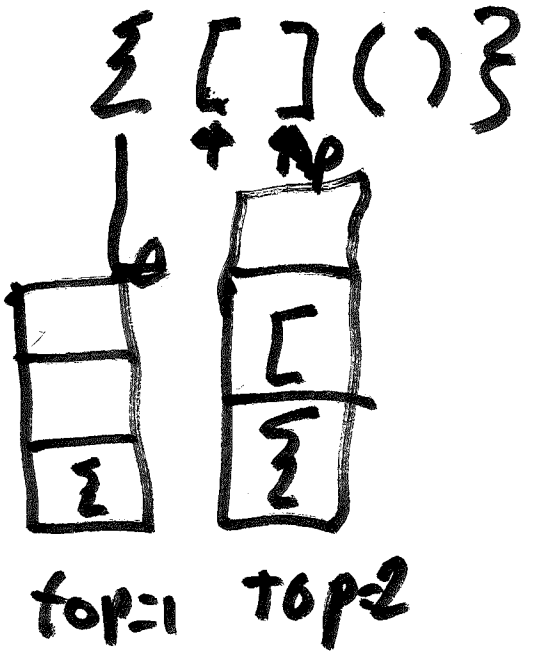
{ [ ) ( ] } Not ok

{ { (}) } Not ok

interleaved

```
if ( [] ) {
   for(  ) {
   }
}
```

```
//Create stack
int len = strlen(s);
char *stk = new char[len];
int top = 0;

char *p = s;
while (*p != '\0') { //iterate over all chars.
    if (*p == '{' //*p == '[' //*p == '('){
        //push into stack.
        stk[top] = *p;
        top++;
    }
    else if (*p == '}' //*p == ']' //*p == ')'){
        if (top == 0) {
            // No matching close.
            return false;
        }
```




```
// there is an element in the
// stack. pop c
char c = stk[top--];
top--;
if ( c == '{' ) {
    if (*p != '}') {
        delete []stk
        return false
    }
    else
}
else if (c == '[') {
    if (*p != ']') {
        delete stk;
        return false
    }
else if ( c == '(') {
    if (*p != ')') {
        delete stk;
        return false
    }
}
```

```
        }
    }
    // We reached the end of string
    // make sure nothing was left
    if (top > 0) {
        delete stk;
        return false;
    }
    delete stk;
    return true;
}
```

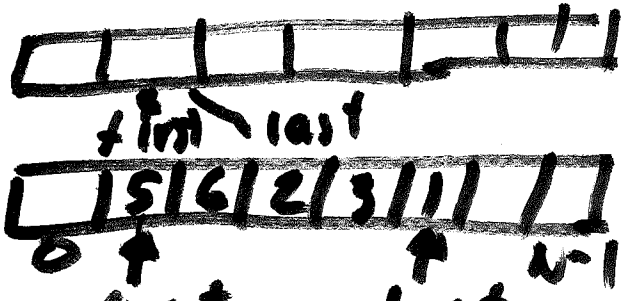
Queues

They are data structures that follow the policy FIFO, that is the next item removed will be the oldest one in the queue.

Implementation of a queue using an array

insert 5
insert 6
insert 2
remove → 5
remove → 6
remove → 2.

5, 6, 2



We use an array and two indexes: first and last.

- We add at the "last" position and remove from "first" position and increment first and last after every operation.
- When last or first reach the end they will start over at 0.
- The maximum number of elements in the queue will be the size of the array (n).

Quiz 4

What is the advantage of using templates vs. using void * for data
(Answer best option)

a) Type checking

b) More clear code

c) Runs faster

d) All of the above.

Implementation:

QueueInt.h

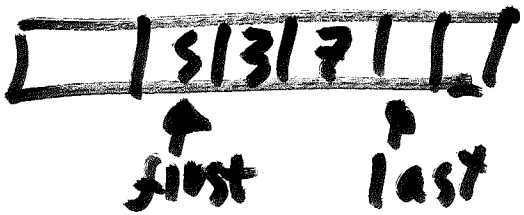
```
class QueueInt {
    int first;
    int last;
    int n; // # of elems in queue
    int maxn;
    int *array;
public:
    QueueInt(int maxn);
    bool enqueue(int val); // returns false
                           // if full
    bool dequeue(int &val); // return false
                            // if empty
   ~QueueInt();
};
```

QueueInt.cpp

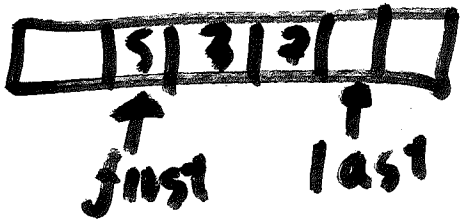
```
QueueInt::QueueInt(int maxn) {
 first = 0;
 last = 0;
 n = 0;
 this->maxn = maxn;
 array = new int[maxn];
}

QueueInt::~QueueInt() {
  delete[] array;
}
```

```
bool
QueueInt::dequeue(int &val){
    //Check if queue is
    //empty
    if(n==0){
      //queue is empty
       return false;
    }
    //Get oldest value
    val=array[first];
    first=(first+1)%maxN;
    n--;
     return true
}
```

```
int val;
Queue.dequeue(val)
//dequeue will
// modify val
// variable passed.
```



```
bool
QueueInt::enqueue(int val) {
    //last is the index of
    //the empty space in
    //queue were we will
    //insert
    if (n == maxn) {
        //no space
        return false;
    }
    //there is space
    array[last] = val;
    last = (last+1) % maxn;   // increment and
                              // wrap around if
                              // necessary
    n++;
    retuin true;
}
```



ResizableArrayInt.h

```
class ResizableArrayInt {
    int maxn; // size of array before
              // having to resize
    int n;    // # of elements used in
              // array
    int *array;
public:
    ResizableArrayInt(int initialSize = 100);
   ~ResizableArrayInt();
    bool setAt(int pos, int val); // set val
                                  // at pos.
    bool getAt(int post, int& val); // rets false
                                    // if pos out
                                    // of bound
    void insertAtEnd(int val);
};
```

ResizableArrayInt.cpp

```
#include "ResizableArrayInt.h"

ResizableArrayInt::ResizableArrayInt(
                        int initialSize) {
    maxn = initialSize;
    n = 0;
    array = new int[initialSize];
}

ResizableArrayInt::~ResizableArrayInt(){
    delete[] array;
}
```

```
void
ResizableArrayInt::InsertAtEnd(int val)
{
  //Check if there is space
  if (n==maxn) {
    // Expand array
    [redacted]
    maxn = 2 * maxn;
    int *newarray = new int[maxn];
    //Copy old array to new array
    for(int i=0; i<n; i++){
       newarray[i] = array[i];
    }
    delete []array;
    array = newarray;
  }
```

```
    // there is space in array
    array[n] = val;
    n++;
}

bool ResizableArray::setAt(int pos, int val){
    //check if pos is between bounds.
    if (pos < 0 || pos >= n){
        return false;
    }
    array[pos] = val;
    return true;
}

bool ResizableArray::getAt(int pos, int &val){
    if (pos < 0 || pos >= n){
        return false;
    }
    val = array[pos];
    return true;
}
```

Analysis

Inserting an item will be most of the time O(1) except whe we have to resize. Resizing is O(n).

For n inserts

cost of insert: n O(1)

Cost of resize: #of resizes * O(n).

#of resize for n elements: O(log n)

maxn

0 100 - m

1 200 - 2m

2 400 - 4m

3 800 - 8m

r 1600 - 16m 2^r = n = maxn = 16m 2^r

#of resize

$n < 16m\,2^r$

$\frac{n}{16m} < 2^r$ ← $\log_2 \frac{n}{16m} < r$

$r = O(\log n)$

Total cost for n iserts = $n\,O(1) + O(\log n)$

$= O(n \log n)$ O(n)

If instead of duplicating maxn

maxn = 2 * maxn

during resize, we increase by a constant.

maxn = maxn + increment;

then the number of resizes for n elements will be:

$\#of\_resize\_if\_increment = O\left(\frac{n}{increment}\right)$

$= O(n)$

So the total cost for n inserts if we increment maxn by a constant is:

$n\,O(1) + O(n)\,O(n) = O(n^2)$

(n inserts; O(n) # resize; O(n) cost of resize)

So it is better to double the size of ~~the array.~~

| Worst case = $O(n)$ when insert resizes | Amortized cost (average cost) for n inserts = $\frac{\text{Cost of n inserts}}{n}$ over for one insert $= \frac{O(n \log n)}{n} = O(\log n)$ |

Quiz 5

What is the total cost of n inserts in a resizable array assuming you double the max size at every resize.

a) $O(n)$
b) $O(n^2)$
c) $O(n \log n)$
d) $O(\log n)$

Summary of cost of resizable array

→ If the initial size is large enough to store all insertions. i.e. $n < max\ n$. No resize needed

insert worst case $O(1)$

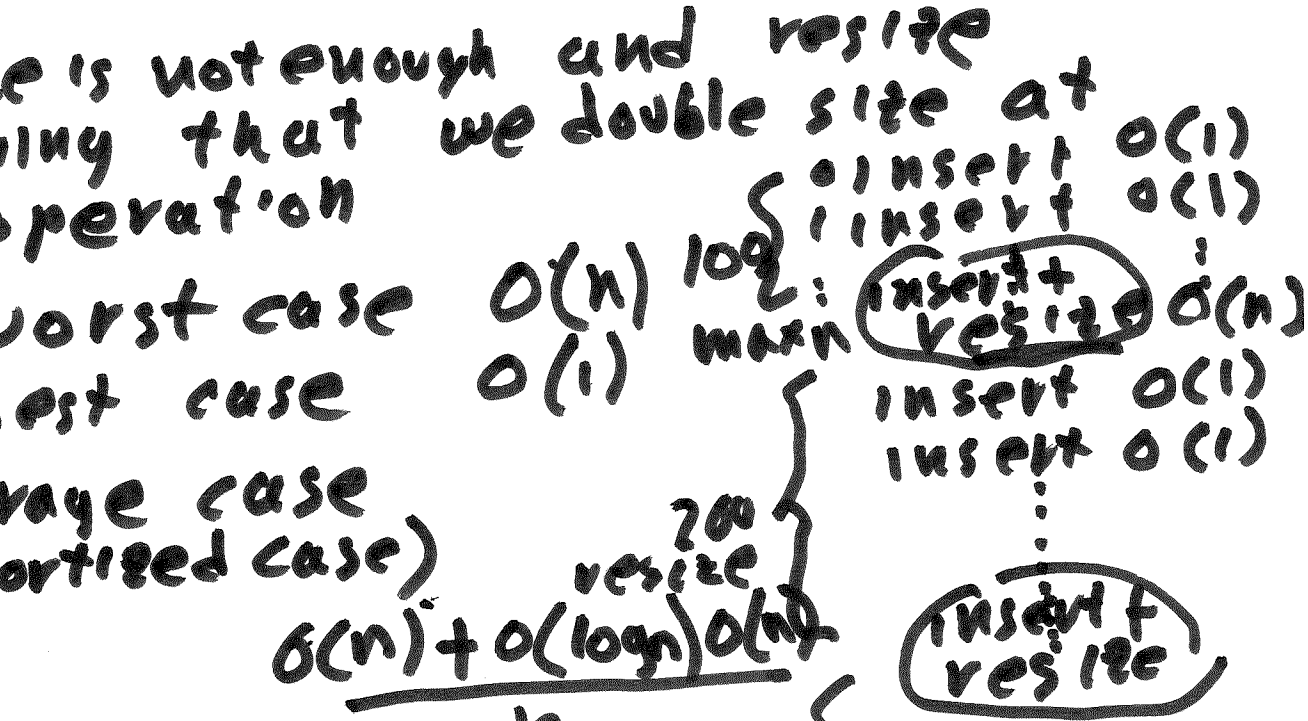
→ If the initial size is not enough and resize is needed, assuming that we double size at every resize operation

insert worst case $O(n)$

insert best case $O(1)$

insert average case (amortized case)

•



$$\frac{O(n) + O(\log n)\, O(n)}{n}$$

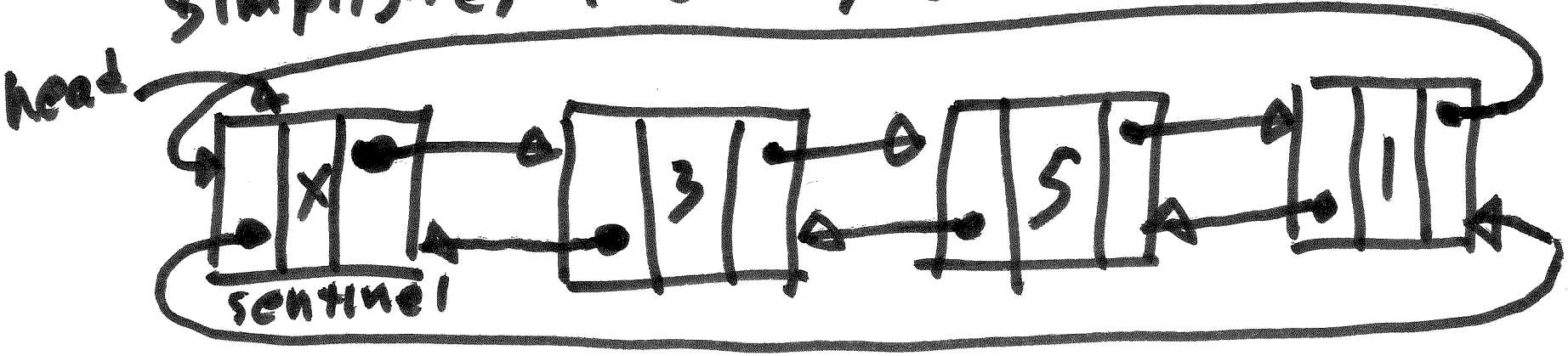
$$= \frac{O(n) + O(n \log n)}{n}$$

$$= O(\log n)$$

Double Linked lists

→ It is a data structure that stores elements in a list where each node has a previous and next pointer.

→ There is a node at the beginning of the list called "sentinel" or "head" that does not store any elements but simplifies the implementation.



double linked lists are popular because

Advantages

→ No resizing is needed like in arrays

insert is O(1) worst case.

InsertFront(val) O(1)
InsertEnd(val) O(1)

→ You can insert at the end in O(1) compared to single linked lists that insertion at end is O(n).

→ Given a pointer to a node, we can insert before or after the node in O(1)

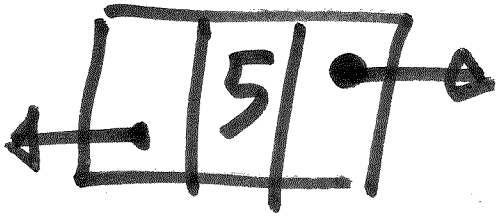
Disadvantage

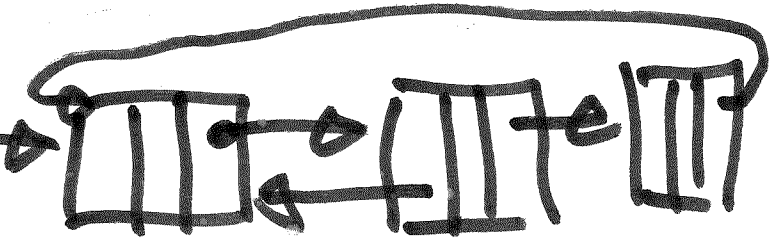
val=getElemAt(i) → Get an element at a position in a double linked list is O(n). In an array this operation is O(1).

DLList.h

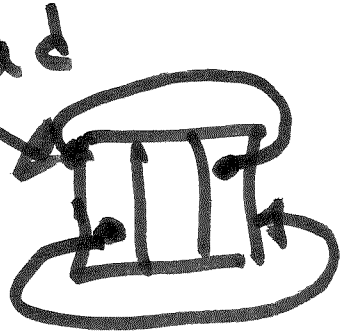
```
class DLLNode {
   int val;
   DLLNode *next;
   DLLNode *prev;
};

class DLList {
   DLLNode *head;
public:
   DLList();
  ~DLList();
   void insertFront(int val);
   bool removeFront(int &val);  // puts val of
                                // in val and
                                // ret false
   void insertEnd(int val);     // if err.
   bool removeEnd(int &val);
```

DLLNode





```
bool IsEmpty();
    void insertAfter(DLLNode *node, int val);
    void insertBefore(DLLNode *node, int val);
    void remove(DLLNode *node);
};
```

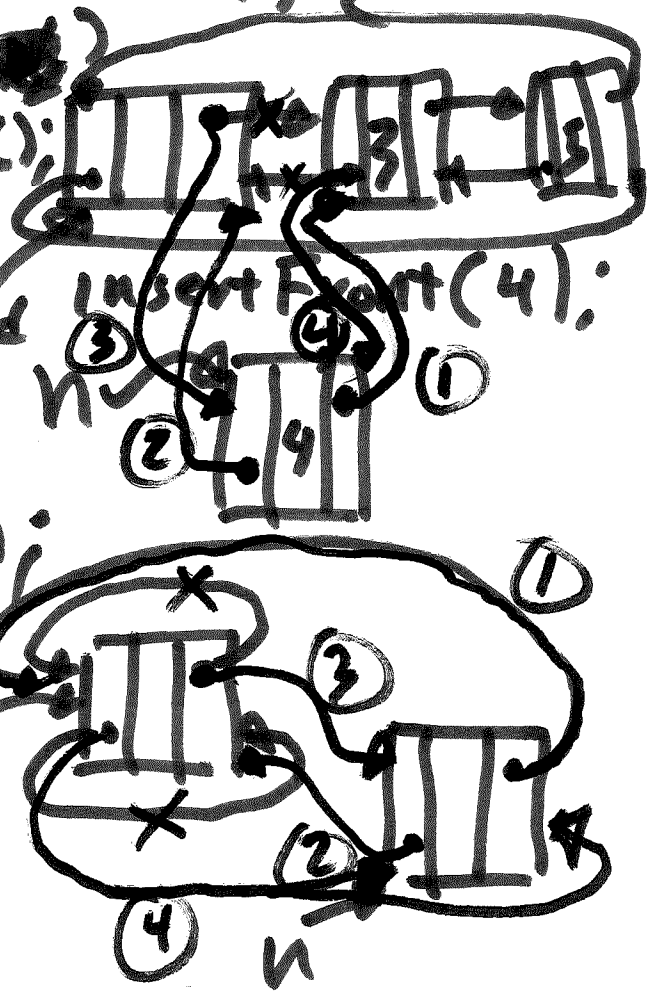


DLList.cpp

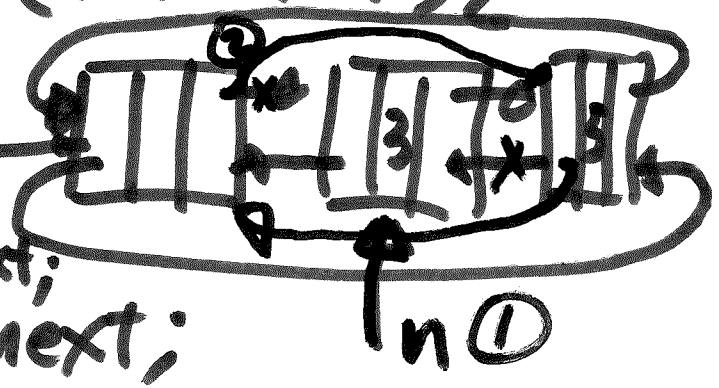
```
#include "DLList.h"

DLList::DLList() {
    head = new DLLNode();
    head->next = head;
    head->previous = head;
}
```

```
void
DLList::insertFront(int val) {
    DLLNode *n = new DLLNode();
(1) n->next = head->next;
(2) n->previous = head;
(3) head->next = n;
(4) n->next->previous = n;
    n->value = val;
}
```

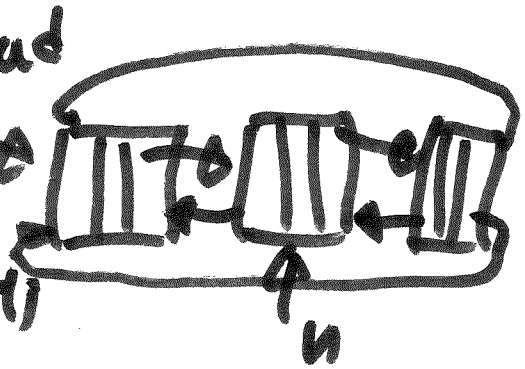
O(1)



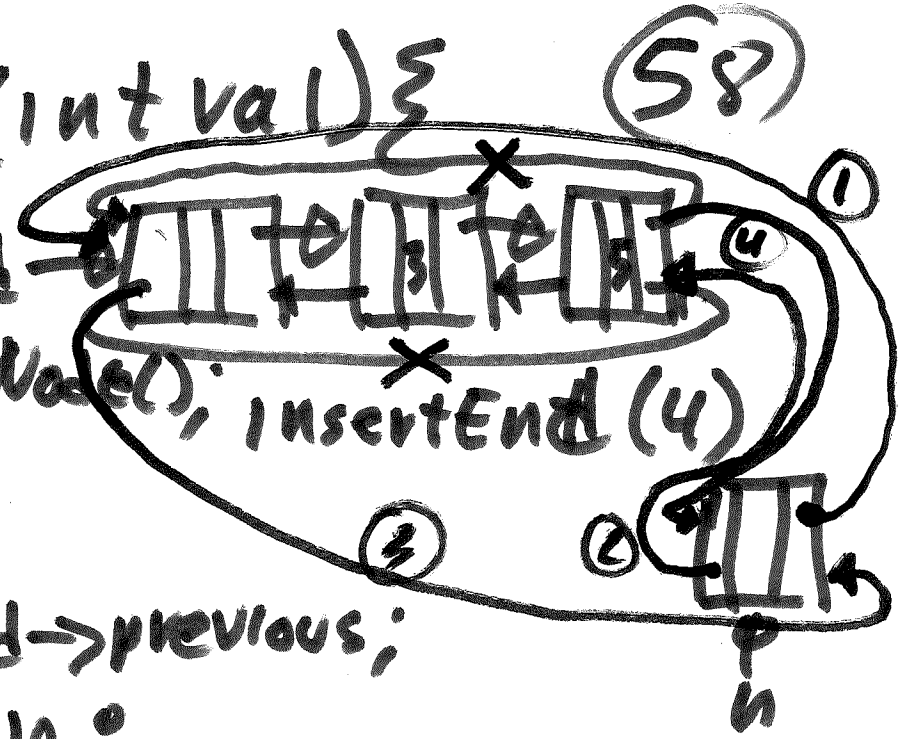
```
bool
DLList::removeFront(int &val){
  if(isEmpty()){
    return false;
  }
① DLLNode *n=head->next;
② head->next = n->next;
③ n->next->previous=head;
  val = n->val;
  delete n;
}
```



```
bool
DLList::isEmpty() {
    return head->next == head;
}

DLList::~DLList() {
    DLLNode *n = head->next;
    while (n != head) {
        //save next pointer
        DLLNode next = n->next;
        delete n;
        n = next;
    }
    delete head;
}
```



```
void
DLList::insertEnd(int val){
  DLLNode *n=new DLLNode(); insertEnd(4)
  n->val=val;
① n->next=head;
② n->previous=head->previous;
③ head->previous=n;
④ n->previous->next=n;
}
```



Quiz 6

~~The following~~

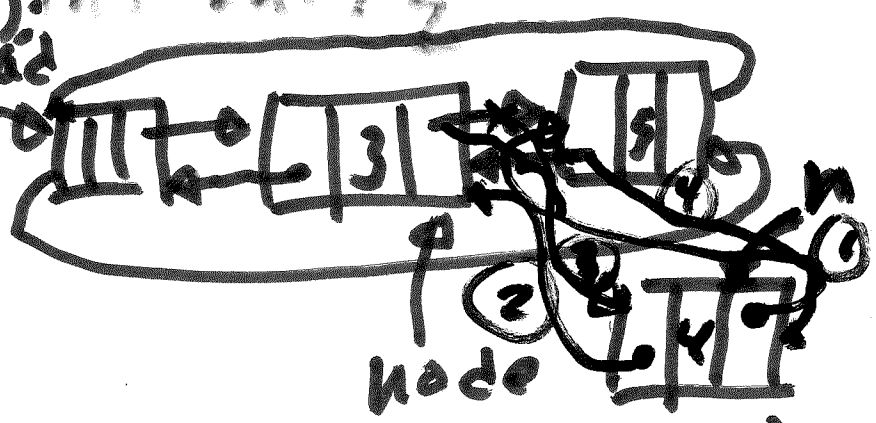
What is the missing line in the following method

```
void
DLList::insertFront(int val) {
  DLLNode *n = new DLLNode();
  n->next = head->next;
  n->previous = head
  [ missing line ]
  n->next->previous = n;
  n->val = val;
}
```

a) `head->previous = n;`
b) `head->next = n;`
c) `n->previous = head`
d) `head->next->previous = n;`

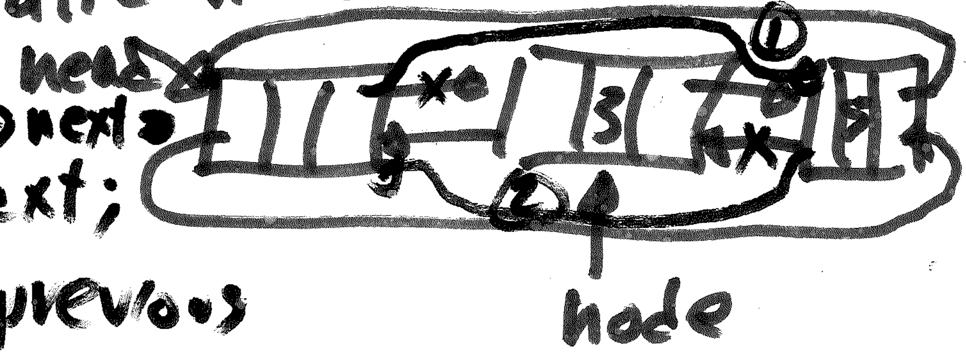
```
void
DLList::insertAfter(DLLNode *node,
                    int val) {
  DLLNode *n = new DLLNode();
  n->val = 4;
  ① n->next = node->next;
  ② n->previous = node;
  ③ node->next = n;
  ④ n->next->previous = n;

}
```

O(1)



insertAfter(node, 4);

```
void
DLList::remove(DLLNode *node){
    //removes node from list.
    //node is a valid node in the list
  ① node->previous->next
        = node->next;
  ② node->next->previous
        = node->previous;
    delete node;
}
```

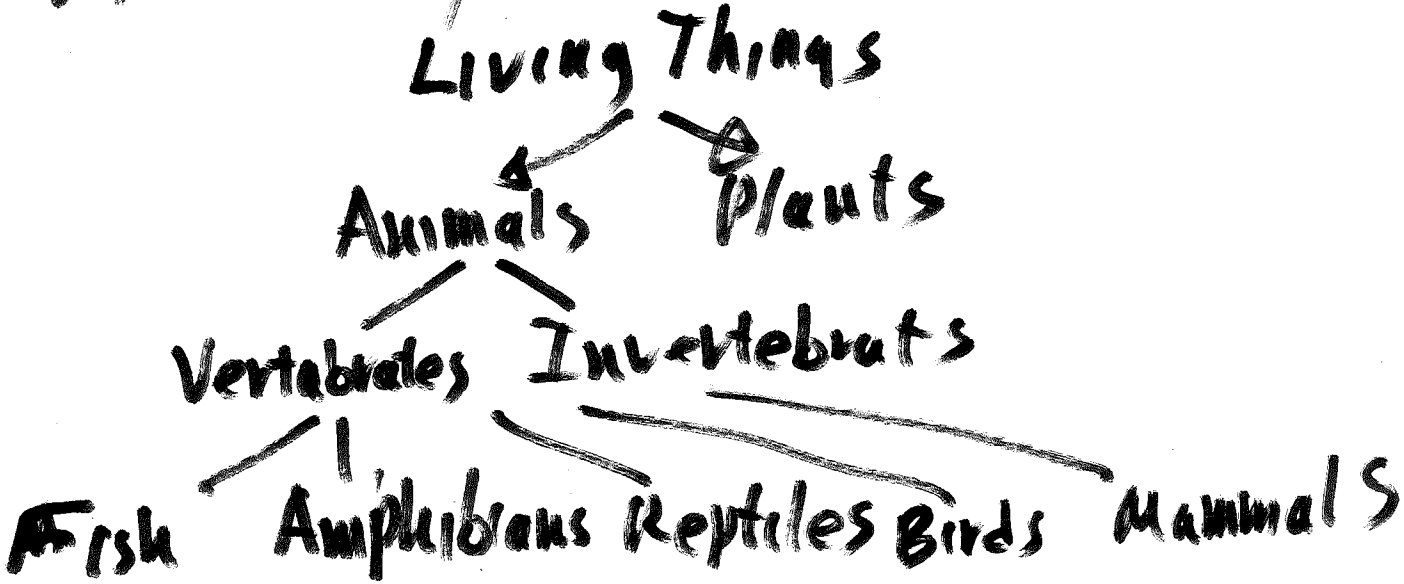


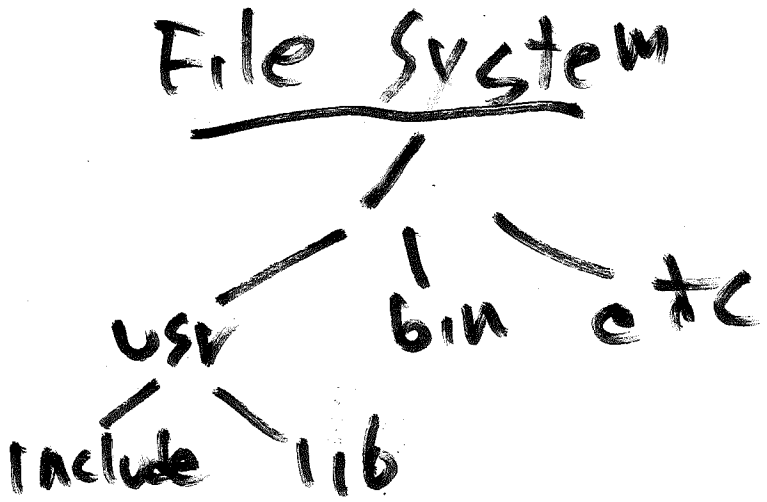
Comparison in array and DLList implement.

	array	DLList
- isEmpty()	O(1)	O(1)
- insertAtEnd	O(1) fixed best; O(n) resizable worst; O(logn) avg amort.	O(1)
- insertFront	O(n)	O(1)
- getValAtPos(ith)	O(1)	O(n)

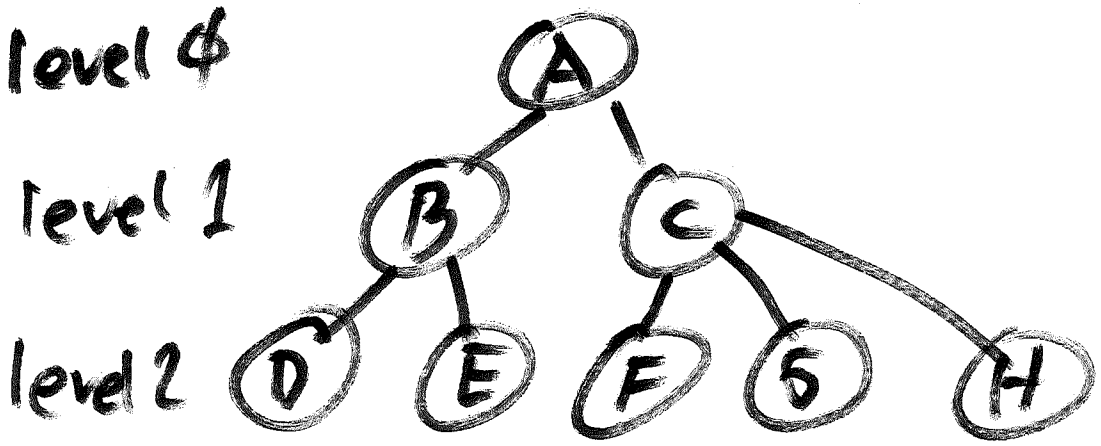
Trees

Data structures used to represent a hierarchy





Vocabulary



A is in the root node

B is the parent of D, E

C is the sibling of B

D, E are children of B

D, E, F, G, H are external nodes

A, B, C are internal nodes

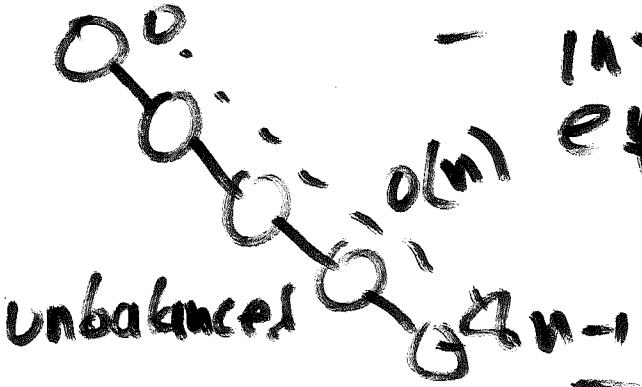
The depth (level) of E is 2.

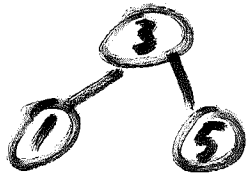
The height of the tree is 3

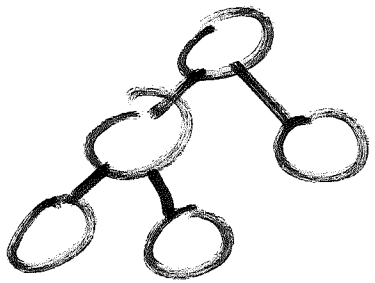
The degree of B is 2 (# of children)

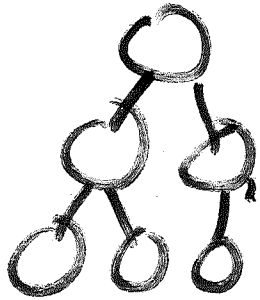
Ordered Tree - The children of each node are ordered.
(there will be different types of ordering).

Binary Tree — ordered tree with all
- internal nodes with a degree equal to 2.



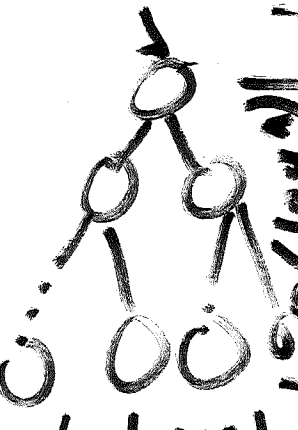






Balanced Tree -

- It is a tree where the distance from the root to any external node is $O(\log n)$ where n is the total number of nodes in the tree.



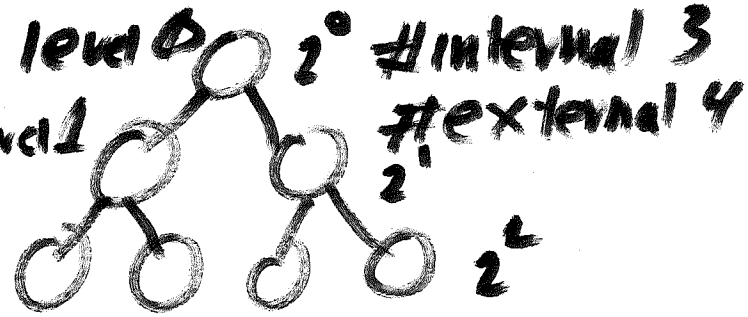
- A balanced tree will allow a traversal from the root to any node to be $O(\log n)$

Properties of Trees

(full tree) # external nodes $\leq$ # internal nodes + 1

nodes at level $i \leq 2^i$ (full tree)

external nodes $\leq 2^{\text{height}-1}$



height = 3

external nodes $\leq 2^{3-1}$

2^2

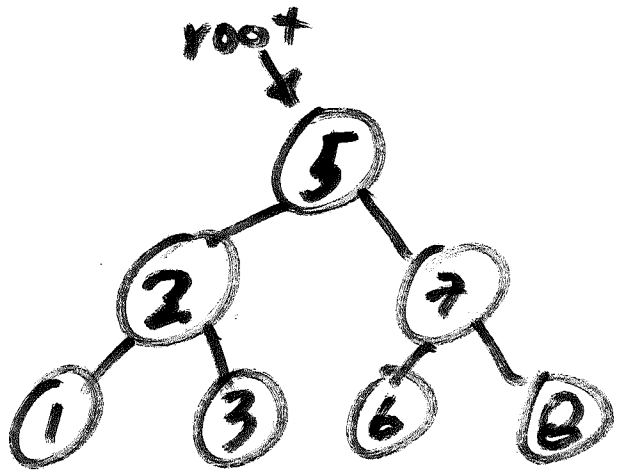
Traversal of trees

- Preorder Traversal
- Postorder Traversal
- Inorder Traversal

Preorder Traversal

Visit parent before children.

```
preorder(v) {
    Visit parent
    for each child of v
        recursively call preorder(c)
}
```



visit ⇒ print value

preorder(root)

5, 2, 1, 3, 7, 6, 8

postorder traversal

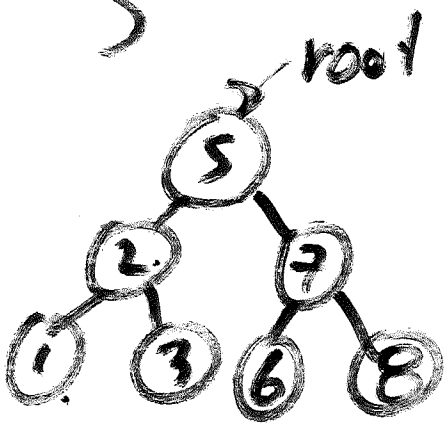
Visit the children before parent.

```
postorder(v) {
    for each child c of v
        recursively call postorder(c)
    visit v
}
```

visit = print value

postorder(root)

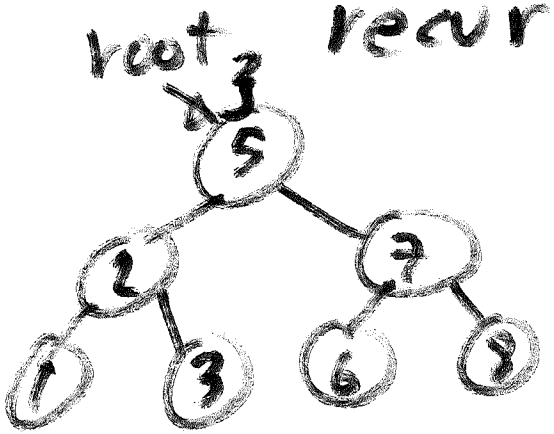
1, 3, 2, 6, 8, 7, 5



inorder traversal

- only for a binary tree
- visit left child, parent, right child

```
inorder(v) {
    recursively call
    inorder(v->left) if there is left child

    visit v

    recursively call inorder(v->right)
    if there is right child
}
```



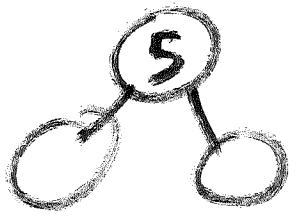
inorder(root)

1, 2, 3, 5, 6, 7, 8

Implementation of a binary tree

BinTree.h

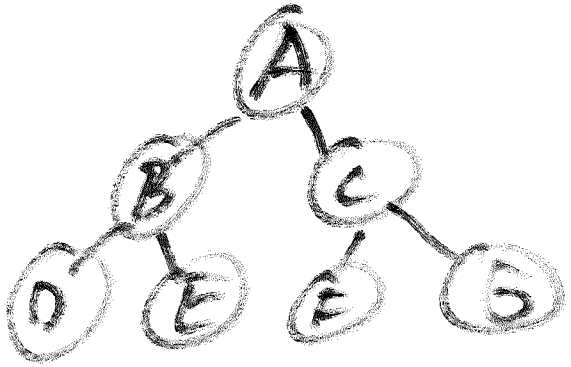
```
struct BTNode {
    int val;
    BTNode *left;
    BTNode *right;
};

class BinTree {
    BTNode *root;
public:
    BinTree();
void insert(int val);
    .
    .
    .
};
```



```
void
BinTree::inOrderPrint(BTNode *node){
    if (node == NULL) return;
    inOrderPrint(node->left);
    printf("%d,", node->val);
    inOrderPrint(node->right);
}

void
BinTree::preOrderPrint(BTNode *node){
    if (node == NULL) return;
    ~~preOrderPrint(node~~
    printf("%d ", node->val);
    preOrderPrint(node->left);
    preOrderPrint(node->right);
}
```

Quiz 7

What is the postorder traversal of the following tree



a) A, B, C, D, E, F, G

b) D, B, E, A, F, C, G

c) D, E, B, F, G, C, A

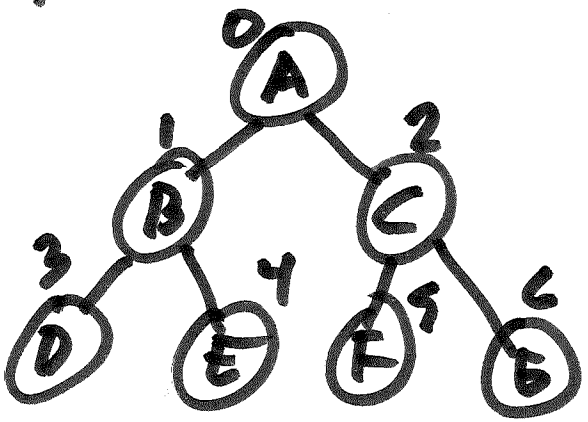
d) G, C, F, E, B, A

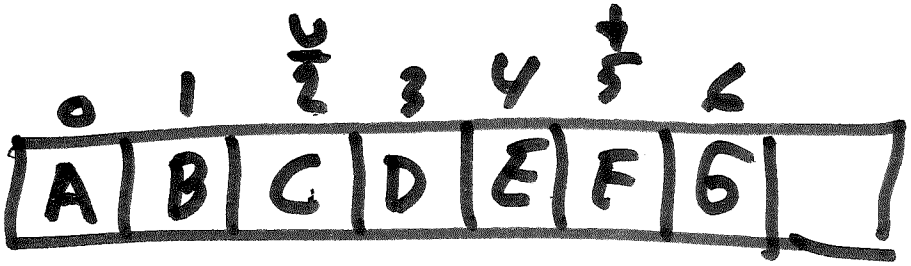
Representation of Binary Trees

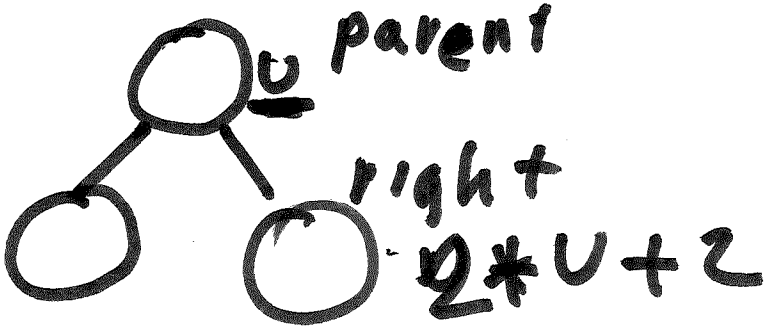
We have seen already a pointer representation of a binary tree.

```
struct BTNode {
    BTNode *left;
    BTNode *right;
    int val;
};
```

We can also represent a binary tree using an array







We assume tree is full. If it is not leave nodes empty if not used.

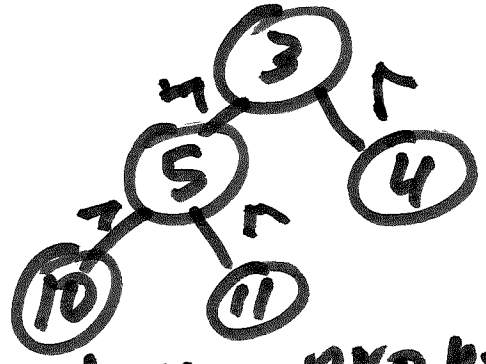
Priority Queue

- A priority queue is a data structure where we can insert items in any order but only the element with the minimum key can be removed.

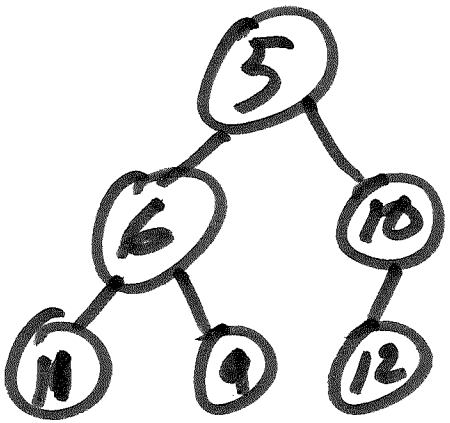
Heaps

- A heap is a data structure that implements a priority queue using a binary tree.

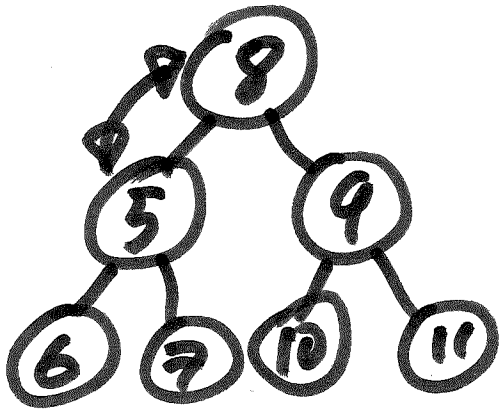
Example:



- It satisfies two properties
  - Order Property: key(parent) ≤ key(child)
  - Structural Property: All levels are full except the last one where all nodes are left filled.



Heap



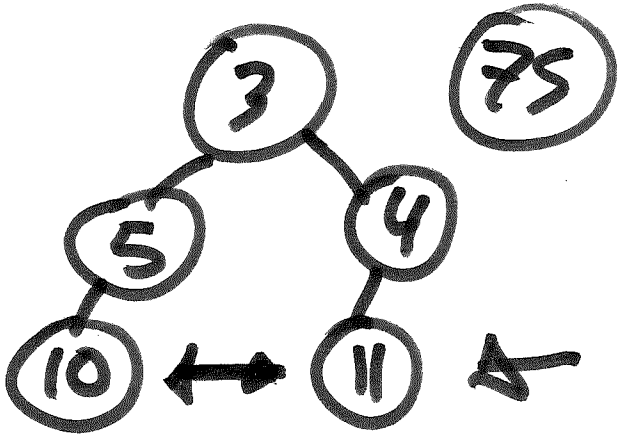
Not a heap

5 < 8

parent

Parent 8 larger than child 5

Doesnot preserve order property

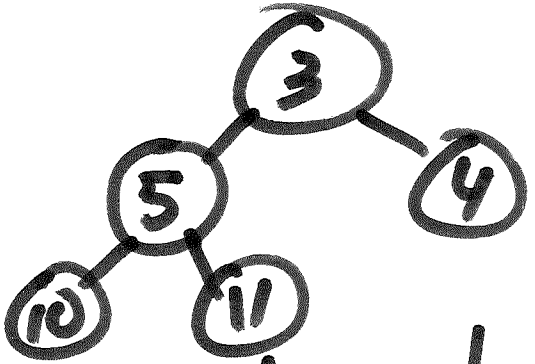


Not a heap

Breaks structural property

Not left filled

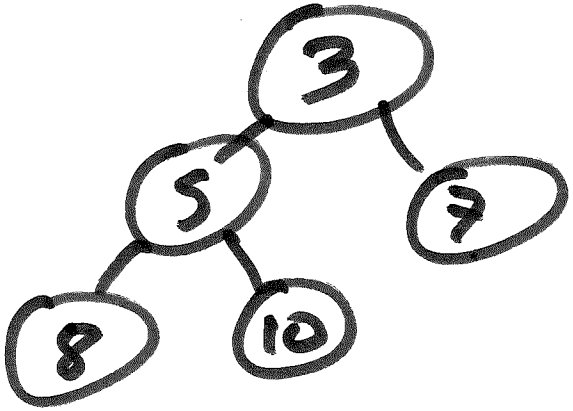
11 has to be child of 5



Heap!

Operations of a Heap

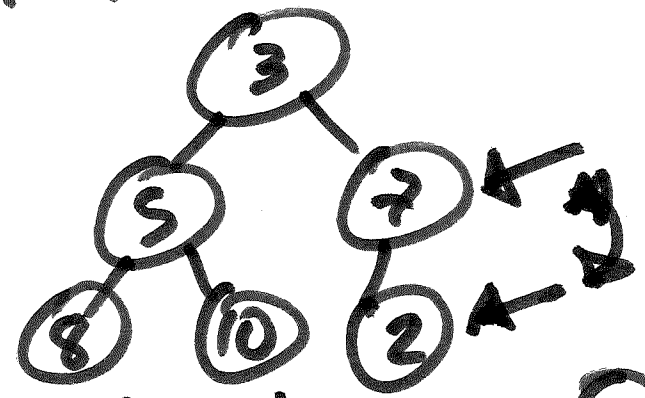
Initial Heap



Insertion

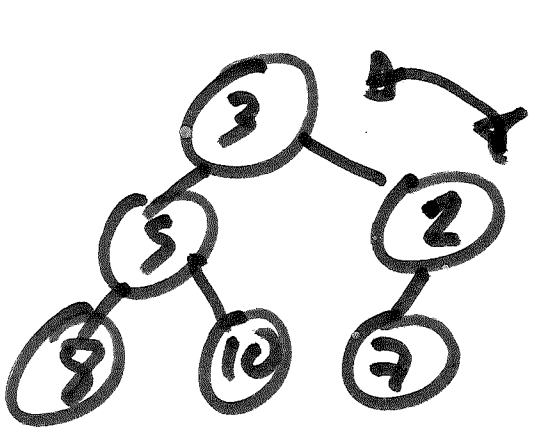
Insert (2)

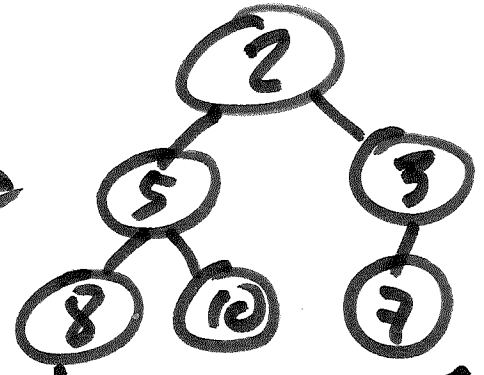
Add key in the first available position



This is not a heap

We do an "up heap" step swapping the child and parent and going up until the heap is ordered or we reach the root.





h
1
2
3

$n \leq 2^h - 1$ $\quad n < 2^h$

total # of sway operations = $h - 1$

height

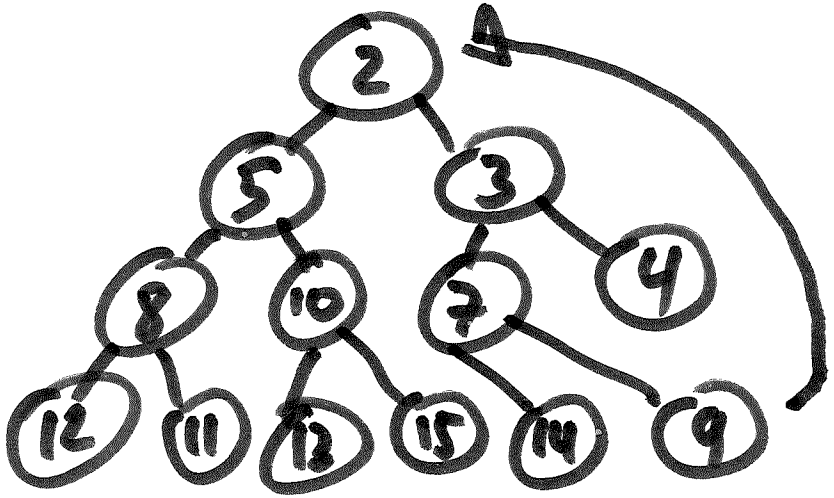
height = $O(\log n)$

→ # of swaps is $O(\log n)$

so insert in a heap takes $O(\log n)$.

Remove Min

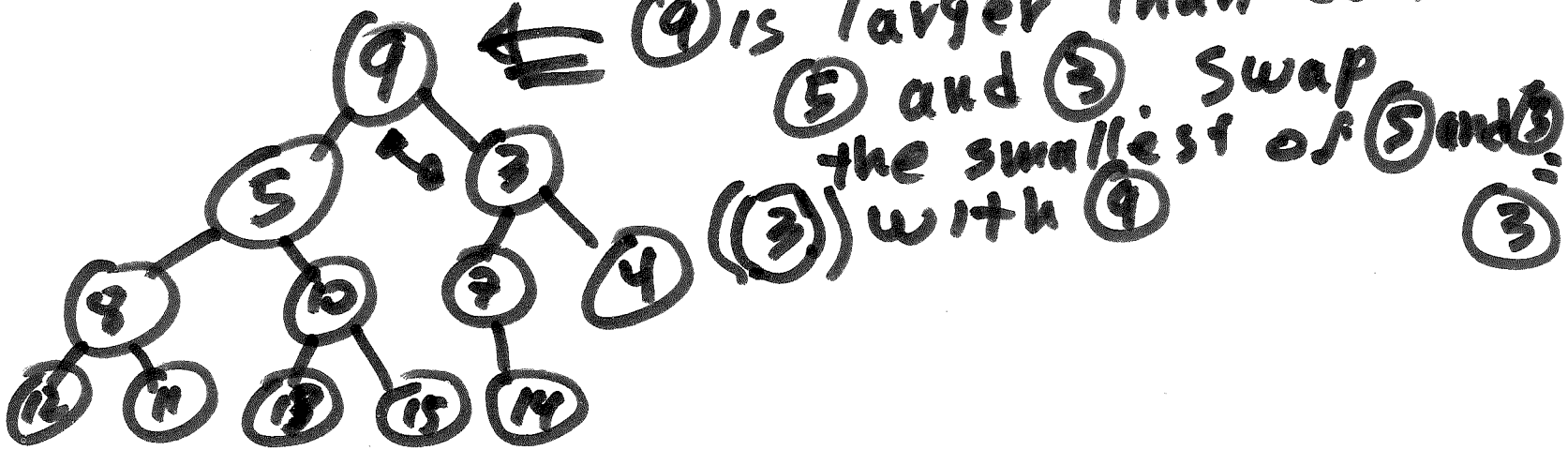
- Remove element at the root
- Removal of root leaves a "hole"
- Replace root with last key of the heap.
- Do a "down heap" step swapping elements parent/children until heap is sorted or we reach bottom.

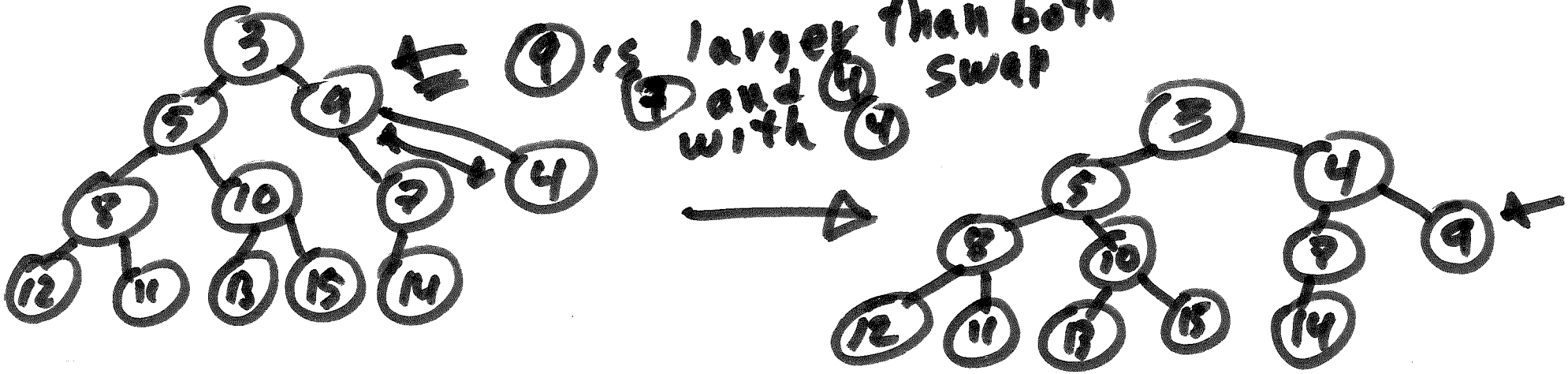


val = removeMin()

val = 2

place 9 instead of 2



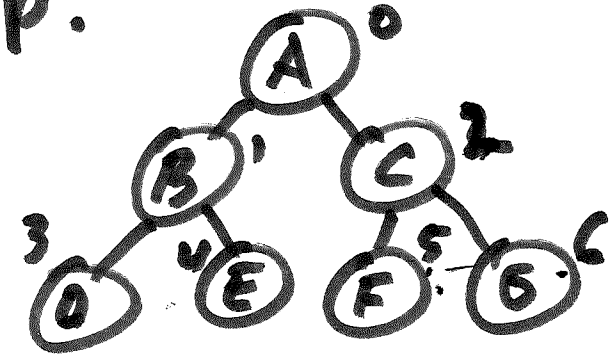


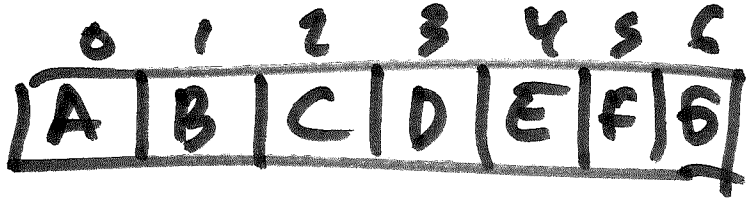
Heap!

Total # of swaps is $h-1$ $h = O(\log n)$

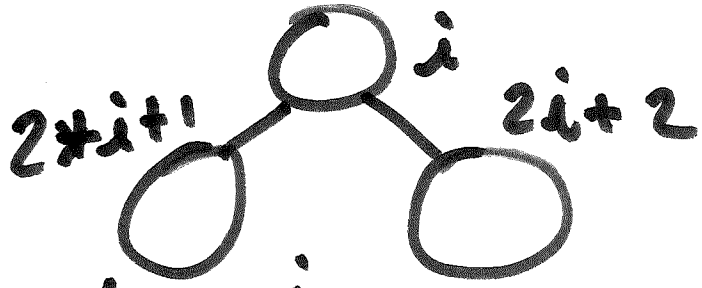
Since h is $O(\log n)$ then the cost of removeMin is $O(\log n)$

We can use the array representation of a binary tree to represent a heap.





parent(i) = (i-1)/2;



// parent is in location i
then leftchild is in 2*i+1 and
rightchild is in 2*i+2

Implementation

Heap of type int.

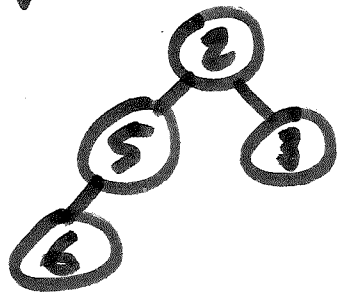
HeapInt.h

```
class HeapInt {
    int n; // current number of elements in heap.
    int nmax; // Maximum number of elements
    int *array; // array that stores heap.
public:
    HeapInt(int maxSize);
    void insert(int key); // Inserts key.
    int removeMin(); // Returns min key
} ~HeapInt();

#define left_child(i)   (2*(i)+1)
#define right_child(i)  (2*(i)+2)
#define iparent(i)      (((i)-1)/2)
```

HeapInt.cc

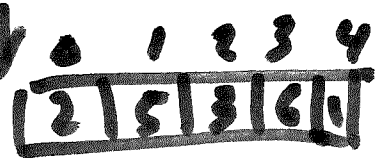
```
HeapInt::HeapInt(int maxSize){
    n = 0;
    nmax = maxSize;
    array = new int[maxSize];
}

HeapInt::~HeapInt(){
    delete[]array
}
```

```
void
HeapInt::insert(int key){
  // inserts key in heap.
  assert(n < nmax); //Make sure there is space
  //add key in the next position available
  array[n] = key;
  n++;
  //Apply up-heap. swap
  // parent and child until
  // swapping is not needed parent
  int child = n-1;
  int parent = parent(child);
  while (child > 0){
    if (array[child] > array[parent]){
      //No need to swap. stop.
      break;
  }
}
```

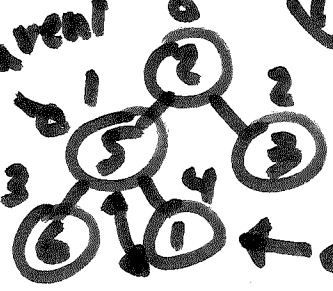


n = 4

insert(1)



n = 5

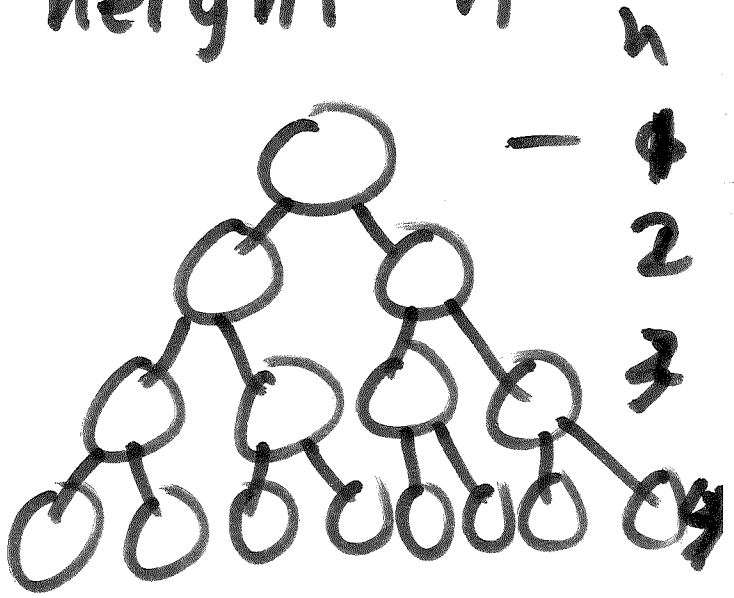


O(height)

O(log n)

Quiz 8

What is the maximum number of keys n_{max} in a heap with height h



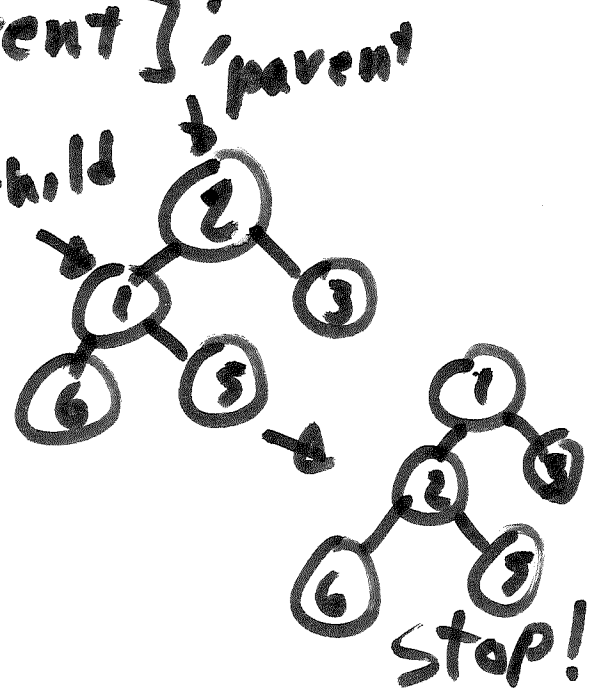
a) $n_{max} = h$

b) $n_{max} = h^2$

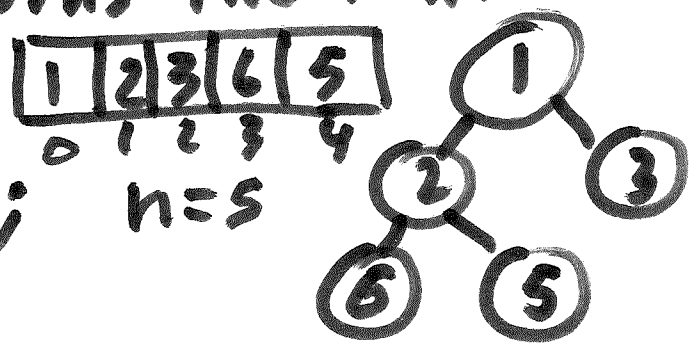
c) $n_{max} = 2^h$

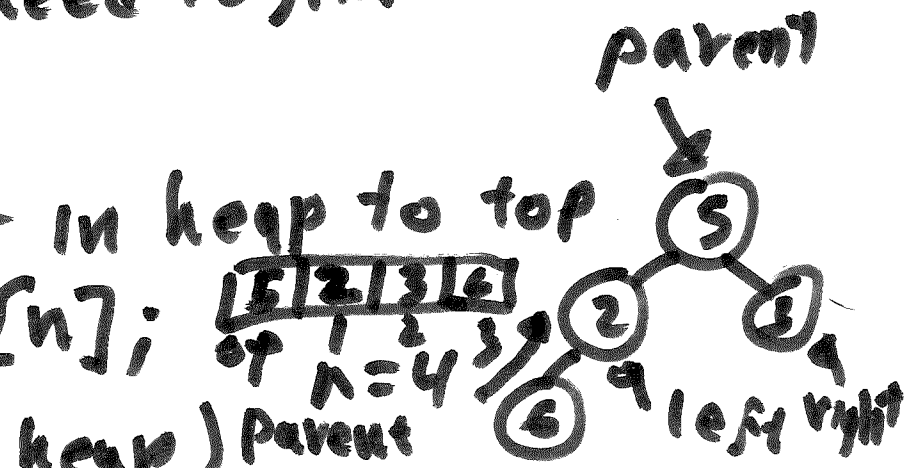
d) $n_{max} = 2^h - 1$

```
// we need to swap parent and child
int tmp = array[child];
array[child] = array[parent];
array[parent] = tmp;
child = parent; // go up
parent = iparent(child);
} // while
} // insert
```



```
int
HeapInt::removeMin() {
  //removes and returns the minimum key.
  assert(n > 0);
  int minKey = array[0];
  n--;
  if (n == 0) {
    //heap empty. No need to fix.
    return minKey;
  }
  //Move last element in heap to top
  array[0] = array[n];

  //Fix heap (down heap) starting at the top
  int parent = 0;
  int left = left_child(parent);
  int right = right_child(parent);
```



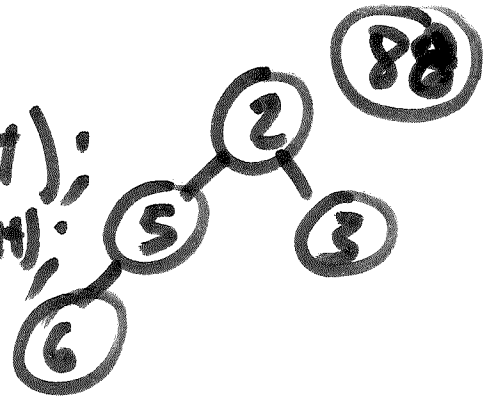


```
while (left < n) { // Repeat
                   // until bottom is
                   // reached.
  //Determine smallest
  //child
  int minChild = left;
  if (right < n && // Make sure there
                   // is a right child
      array[right] < array[left]) {
    minChild = right; // right child is
                      minimum.
  }
  //check if we need to swap
  if(array [parent] < array[minChild]){
    //No need to swap.
    break;
  }
  // we need to swap parent and minChild
  int tmp = array [minchild];
  array [minChild] = array [parent];
  array [parent] = tmp;
```

Bounded by height of the heap that is less than log(n)

O(log n)

```
        //go down
        parent = minchild;
        left = left_child(parent);
        right = right_child(parent);
    } //while

    // heap has been fixed
    return minkey;
}
```



Heap insert — $O(\log n)$

Heap removeMin — $O(\log n)$.

HeapSort

- It is a sorting algorithm that uses a Heap.
- It runs in $O(n \log n)$.

```
#include "HeapInt.h"
void heapSort(int *array, int n){
    // sorts array with n elements
    // in ascending order using heapsort.
    HeapInt heap(n);
    for (int i=0; i<n; i++){   //insert elements
n*log n     heap.insert(array[i]);   //in heap.
    }
    //Now put elements back into array from heap
    //using removeMin
n*log n for (int i=0; i<n; i++){
removeMin     array[i]= heap.removeMin();
    }
}
```

Time Complexity of Heapsort

n - heapInserts = $n\,O(\log(n))$

$O(\log n)$ = $O(n \log n)$

n - removeMin ops. = $n\,O(\log n)$

$O(\log n)$ = $O(n \log n)$

Total time is $O(n \log n) + O(n \log n)$

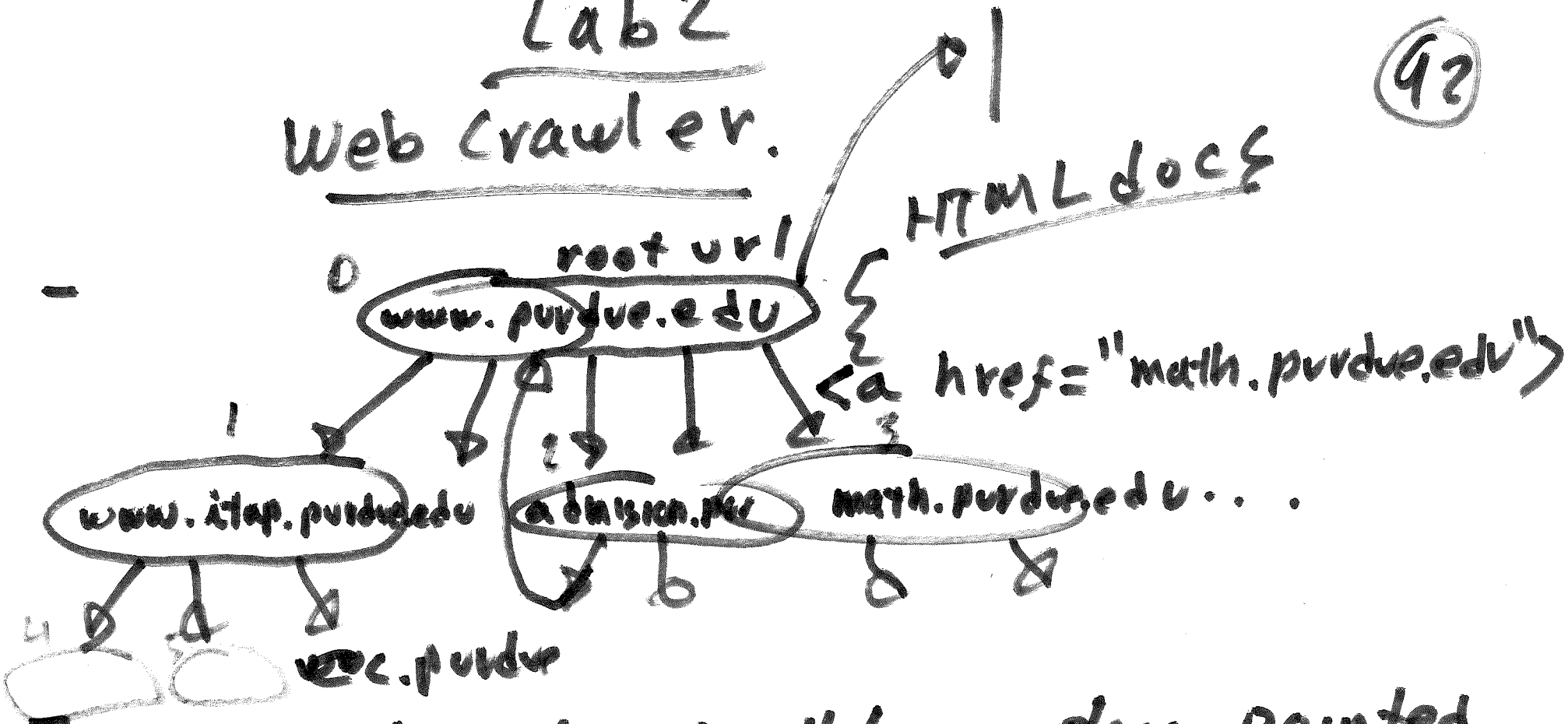
= $O(n \log n)$.

worst case!

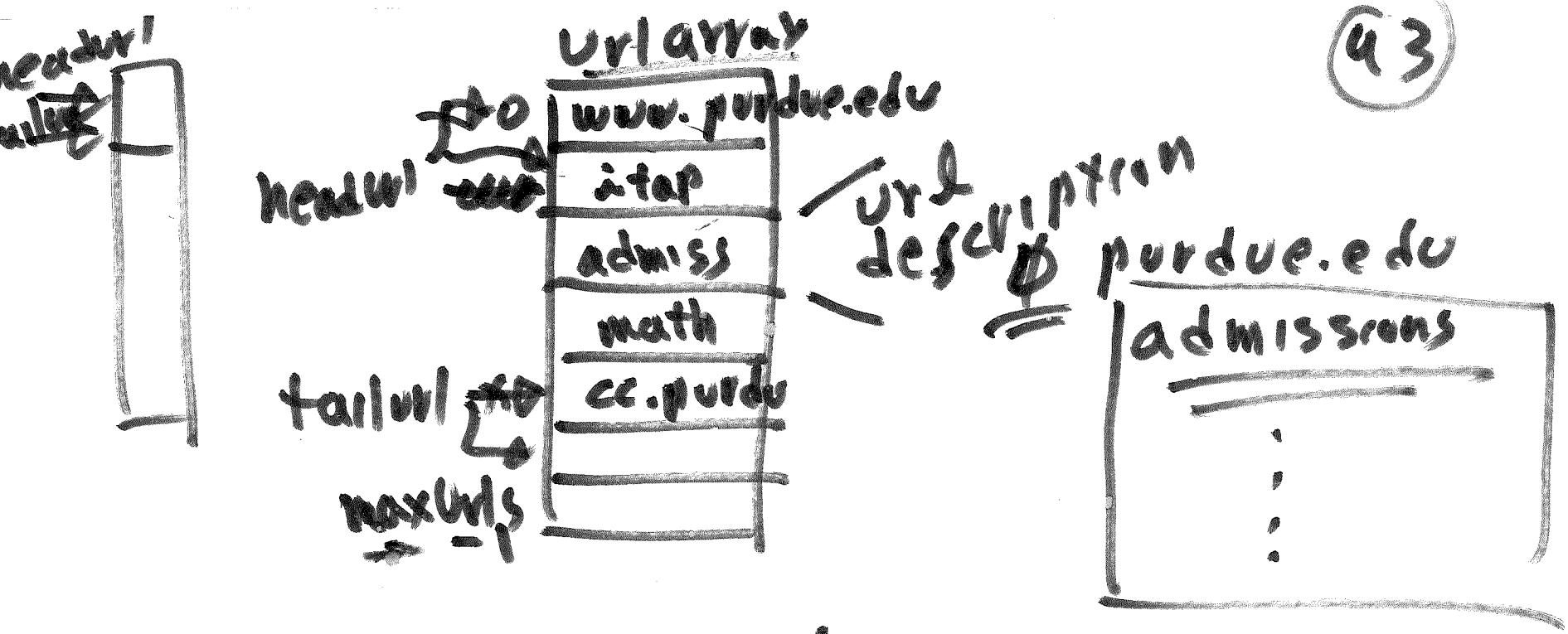
- The problem with the above implementation is that we need extra space in the heap to store array. This implementation is not an "in place" sorting implement.
- We could modify the implementation to use the same array to store the heap but will leave it to you to do at home.
- If is possible to make heapsort to be an in-place algorithm.

Lab2

Web Crawler.



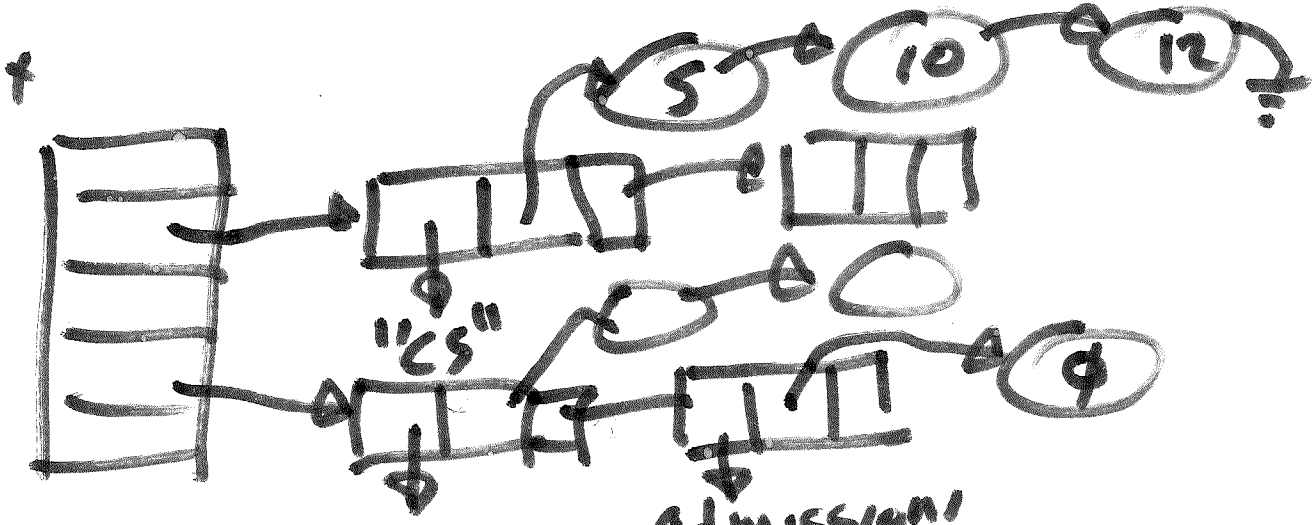
Web crawl will find all/max. docs pointed directly or indirectly from root url.



wordToURLRecordList

Dictionary that maps a word to a urlIndex list

word → list of url indexes



Quiz 9

In heap Sort, if instead of we use an array where insert inserts at the end and removeMin finds the minimum and shifts elements, what is the complexity of sorting

a) $O(n)$
b) $O(n^2)$
c) $O(n \log n)$
d) $O(n^3)$

Dictionaries

It is an abstract model of a data base. Given a key, it finds the data that corresponds to the key.

Methods:

Data findElement(key)
insertElement(key, data)
removeElement(key).

We can implement a dictionary using:

- Hashtable
- Binary Search Dictionary ✓
- Array
- Linked list

Later on also we will cover

- AVL Tree

Binary Search Dictionary

- It uses an array to store entries with key, data.

- The array is sorted by the key.
- The lookup uses binary search.

Binary Search

- Narrow down the range of the search by half at every iteration.
- similar to
  - High-low game.

0	1	2	3	4	5	6	7	8	9	10	11
2	4	7	8	9	12	17	19	22	25	28	37
					↑ mid						↑ high = 11

find(19)

low = 0

$mid = \frac{low + high}{2} = \frac{11+0}{2} = 5$

low = 6

$mid = \frac{6+11}{2} = \frac{17}{2} = 8$

mid = 8

low = 6

high = 7

$mid = \frac{6+7}{2} = 6$

low = 7 high = 7

$mid = \frac{7+7}{2} = 7$ Found!

Time Complexity

At every step we divide the range by half

iteration	range
0	n
1	n/2
2	n/4
⋮	⋮
x	$n/2^x$

we stop when range = 1 → $\frac{n}{2^x} = 1$

$$\frac{n}{2^i} = 1$$

$$n = 2^i$$

$$\log_2 n = i$$

iterations = $O(\log n)$

Binary Search Find takes $O(\log n)$

Quiz 10

What is the time complexity of find(key) using binary search

a) O(1)
b) O(n)
c) O(log n)
d) O(n log n)

Implementation of Binary Search Table

BinSearchTable.h

```
struct BinSearchEnt {
    char * key;
    void * data;
};

class BinSearchTable {
    BinSearchEnt *array;
    int n;
    int nmax;
    bool isSorted;
public:
    BinSearchTable();
    void insert(char *key, void *data);
    void * lookup(char *key);
    ~BinSearchTable();
};
void sortTable(); // It can use any sorting algorithm.
```

```
BinSearchEnt() {
    key = NULL;
    data = NULL;
}
BinSearchEnt(char *key, void *data) {
    thiskey = strdup(key)
    thisdata = data;
}
~BinSearchEnt() {
    if (key != NULL) {
        free(key);
    }
}
```

not needed.

BinSearchTable.cc

```
BinSearchTable::BinSearchTable() {
    n = 0
    nmax = 100;
    array = new BinSearchEnt [nmax];
    isSorted = false;
}

void
BinSearchTable::insert(char *key, void *data){
    //check if there is space.
    if( n == nmax) {
        //No space. Enlarge table
        BinSearchEnt * newarray;
        nmax = 2 * nmax;
        newarray = new BinSearchEnt[nmax];
        for (int i=0; i<n; i++) {
            newarray[i] = array[i];
        }
```

```
        delete [] array;
        array = new array;
    } // if

    // There is space in array
    array[n].key = strdup(key);
    array[n].data = data;
    n++;
    isSorted = false; // We no longer know if
                      // array is sorted.
}
```

```
void *
BinSearchTable::lookup(char *key){
    // Make sure table is sorted.
    if(!isSorted){
        sortTable();
        isSorted = true;
    }
    //Use binary search
    int low = 0;
    int high = n-1;
    while (high >= low) {
        int mid = (high + low)/2;
        int res = strcmp(key, array[mid].key);
                         "kiwi"   "banana" >0
        if (res > 0) {
            low = mid + 1;
        } else if (res < 0) {
            high = mid - 1;
        } else { // key found
            return array[mid].data;
        }
    }
    return NULL; //key does not exist
}
```

low
0 apple
mid
1 banana
2 kiwi
3 orange
4th lookup(kiwi)
high:
low=0 high=3
mid = 0+3/2 = 1

Time Complexity

Bin Search Table insert

worst case O(n)

best case O(1) # expansions

average case (amortized case) O(log n) × O(n) + (n − log n) × O(1) = O(n) = O(log n)

cost of 1 insertion averaged ^ amount × insertions ^

lookup

worst case O(log n) (asuming table is sorted)

worst case (asume table is unsorted) O(n log n)

However binary search tables are used in cases where tables are "Read Only", that is we create the table once with n insertions and then we do m lookups where $m >> n$.

n { insert, insert, ⋮ }

m { lookup → sort, lookup, ⋮ }

amortized case

Cost

n insertions = $n\,O(\log n)$

sorting = $O(n \log n)$

m lookups = $m\,O(\log n)$

Total: $O(n \log n) + O(n \log n) + O(m \log n)$

$= O(n \log n) + O(m \log n)$

If $m >> n$ then Total = $O(m \log n)$

Quiz 11 =

In a binary search table that has resizing. If we do a lookup after every insertion what will be the cost ~~if we sort before a lookup for~~ for n insertions and lookups.

1 insert
2 lookup
3 insert
4 lookup
⋮
n

a) $O(n)$
b) $O(n^2)$
c) $O(n^2 \log n)$
d) $O(n^3)$

Todo:
→ destructor in table
→ time complexity.

Midterm Review

- Math Review
- Types of Running time — worst case, best case, Average case
- Asymptotic Analysis and big-Oh notation $O(g(n))$ if $f(n) \leq c\, g(n)$
- Stacks (arrays) push, pop, LIFO
- Queues (arrays) → (Circular queue) — enqueue, dequeue, FIFO
- Double linked lists.
- Arrays vs Linked Lists

		Avg	Worst
Hash Tables —	insert	O(1)	O(n)
	lookup	O(1)	O(n)

Requirements or Properties that make insert, lookup Avg time O(1)

1. — Hash function has to be Evenly Distributed
2. — Number M of buckets $\geq n$

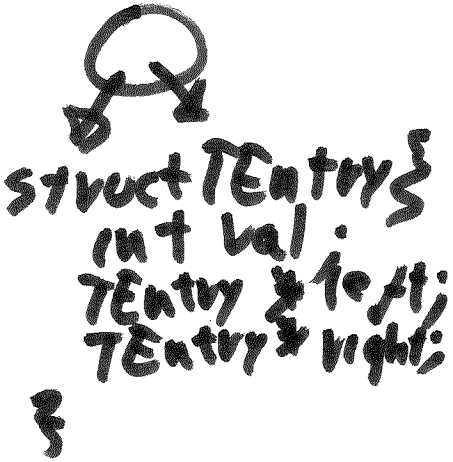
- Implementation

- Resizable array
  - Double size in every expansion
  - Cost of insertion:

Best Case	Worst Case	Avg Case (amortized)
O(1)	O(n)	O(log n)

  - Implementation

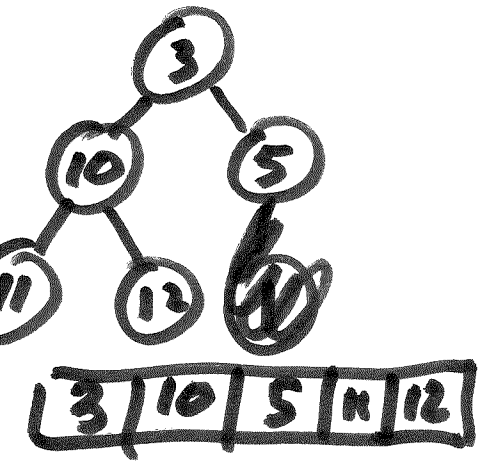
- Trees
  - Traversal
    - Preorder
    - Postorder
    - Inorder
  - Tree representation
    - Pointers left, right



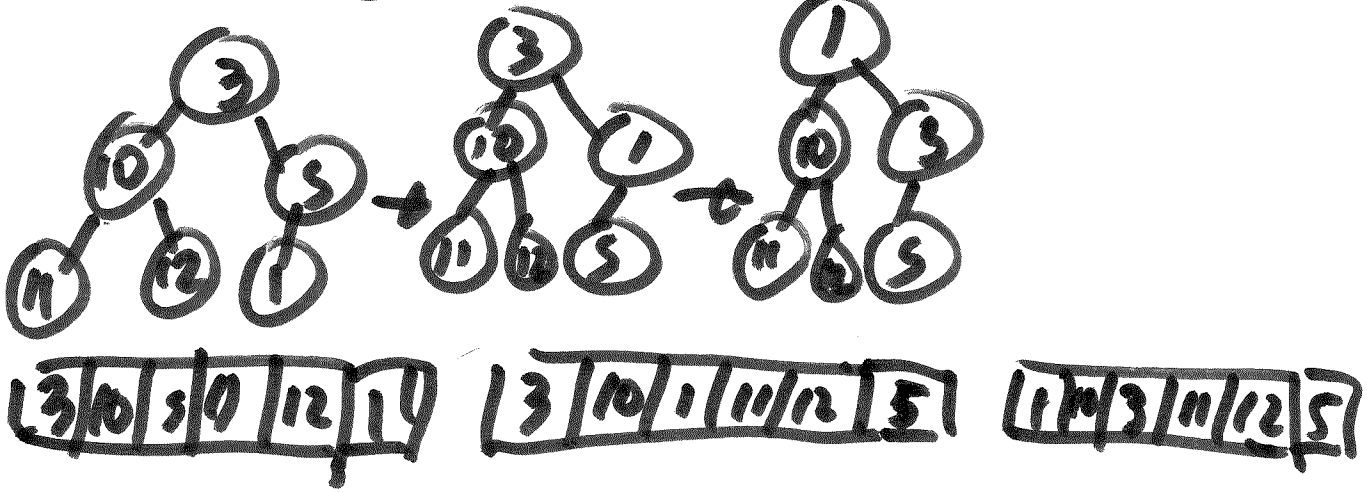
    - Array representation

- Heap data structure
  + Implementation of priority queue
  + Two operations
    - insert
    - removeMin
  + How to perform insert, removeMin in heap
  + Implementation using an array
  + Analysis

	worst case
insert	O(log n)
removeMin	O(log n)



insert 1



- Heapsort
  - worst case - $O(n \log n)$
  - implementation

- Binary Search Dictionary
  - implementation

	best case	worst case	Avg. in lks.
- lookup	$O(\log n)$ no sorting needed.	$O(n \log n)$ (if sorting needed)	
- insertion	$O(1)$	$O(n)$	$O(\log n)$

- Study templates.

To study

- Class Notes (1)
- Book (4)
- Labs (3)
- Homework (2)