COMP202
Complexity of Algorithms

Some data structures (linked lists, binary search trees, heaps)

Reading materials: Chapters 6, 10.2, 10.4, 12 in CLRS.

Learning Outcomes

At the conclusion of this set of lecture notes, you should:

1. Recall the "binary search" method on sorted arrays.
2. Comprehend how binary search trees are used to mimic the binary search method for arrays.
3. Understand priority queues and their implementations via heaps.
4. Understand the HeapSort algorithm.

- Algorithmic computations require data (information) upon which to operate.

The speed of algorithmic computations is (partially) dependent on efficient use of this data.

Data structures are specific methods for storing and accessing information.

Data Structures

- Algorithmic computations require data (information) upon which to operate.

The speed of algorithmic computations is (partially) dependent on efficient use of this data.

Data structures are specific methods for storing and accessing information.

We study different kinds of data structures as one kind may be more appropriate than another depending upon the *type* of data stored, and *how* we need to use it for a particular algorithm.

Data Structures (cont.)

Many modern high-level programming languages, such as C++, Java, and Perl, have implemented various data structures in some format.

For specialized data structures that aren't implemented in these programming languages, many can typically be constructed using the more general features of the programming language (e.g. pointers in C++, references in Perl, etc.).

Data Structures: Arrays

Arrays are available in some manner in every high-level programming language. You have encountered this data structure in some programming course(s) before.

Data Structures: Arrays

Arrays are available in some manner in every high-level programming language. You have encountered this data structure in some programming course(s) before.

- Abstractly, an *array* is simply a collection of items, each of which has its own unique "index" used to access this data. So if A is an array having n elements, we (typically) access its members as $A[0], A[1], \ldots, A[n-1]$.

Data Structures: Arrays (cont.)

One possible difficulty using arrays in real-life programs is that we must (usually) specify the *size* of the array *as it is created*. If we later need to store more items than the array can currently hold, then we (usually) have to create a new array and copy the old array elements into the newly created array.

Data Structures: Arrays (cont.)

One possible difficulty using arrays in real-life programs is that we must (usually) specify the *size* of the array *as it is created*. If we later need to store more items than the array can currently hold, then we (usually) have to create a new array and copy the old array elements into the newly created array.

So we have quick access to individual array members via indexing, but we need to know how many elements we want to store *beforehand*.

Data Structures: Arrays (cont.)

One possible difficulty using arrays in real-life programs is that we must (usually) specify the *size* of the array *as it is created*. If we later need to store more items than the array can currently hold, then we (usually) have to create a new array and copy the old array elements into the newly created array.

So we have quick access to individual array members via indexing, but we need to know how many elements we want to store *beforehand*.

Another possible disadvantage is that it is difficult to insert new items into the "middle" of an array. To do so, we have to copy (shift) items to make space for the new item in the desired position.

Data Structures: Arrays (cont.)

One possible difficulty using arrays in real-life programs is that we must (usually) specify the *size* of the array *as it is created*. If we later need to store more items than the array can currently hold, then we (usually) have to create a new array and copy the old array elements into the newly created array.

So we have quick access to individual array members via indexing, but we need to know how many elements we want to store *beforehand*.

Another possible disadvantage is that it is difficult to insert new items into the "middle" of an array. To do so, we have to copy (shift) items to make space for the new item in the desired position.

So it can be difficult (i.e. *time-consuming*) to maintain data that is *sorted* by using an array.

You have previously seen linked lists in COMP 108.

In contrast to arrays, data can be easily inserted *anywhere* in a linked list by inserting a new node into the list and reassigning pointers.

A list abstract data type (ADT) supports: *referring*, *update* (both *insert* and *delete*) as well as *searching* methods.

Data Structures: Linked Lists

You have previously seen linked lists in COMP 108.

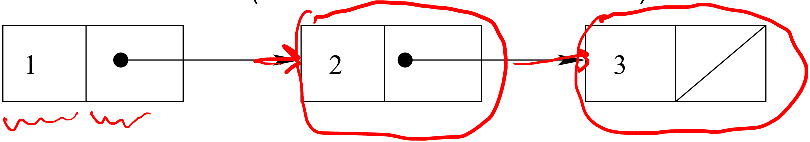
In contrast to arrays, data can be easily inserted *anywhere* in a linked list by inserting a new node into the list and reassigning pointers.

A list abstract data type (ADT) supports: *referring*, *update* (both *insert* and *delete*) as well as *searching* methods.

We can implement the list ADT as either a *singly-*, or *doubly-* linked list.

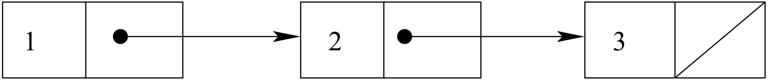
Linked List

- A *node* in a *singly*-linked list stores a *next* link pointing to next element in list (**null** if element is last element).

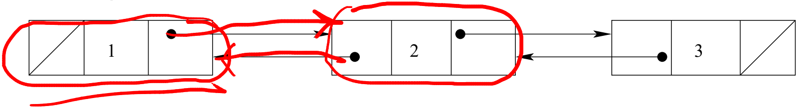


Linked List

- A *node* in a *singly*-linked list stores a *next* link pointing to next element in list (**null** if element is last element).



- A *node* in a *doubly*-linked list stores two links: a *next* link, pointing to the next element in list, and a *prev* link, pointing to the previous element in the list.



From now on, we will concentrate on doubly-linked lists.

List ADT: Update methods

A list ADT supports the following *update* methods:

- *replaceElement(p,e)*: *p* - position, *e* - element.
- *swapElements(p,q)*: *p*, *q* - positions.
- *insertFirst(e)*: *e* - element.
- *insertLast(e)*: *e* - element.
- *insertBefore(p,e)*: *p* - position, *e* - element.
- *insertAfter(p,e)*: *p* - position, *e* - element.
- *remove(p)*: *p* - position.

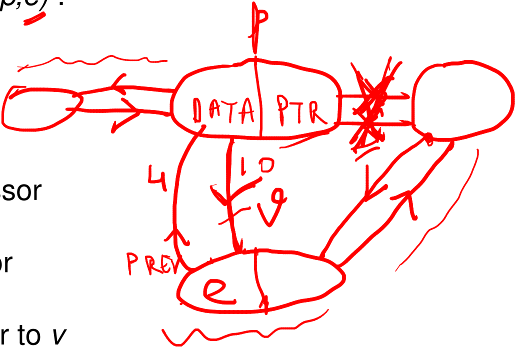
EXPENSIVE FOR ARRAYS.

List update: Element insertion

Pseudo-code for *insertAfter(p,e)* :

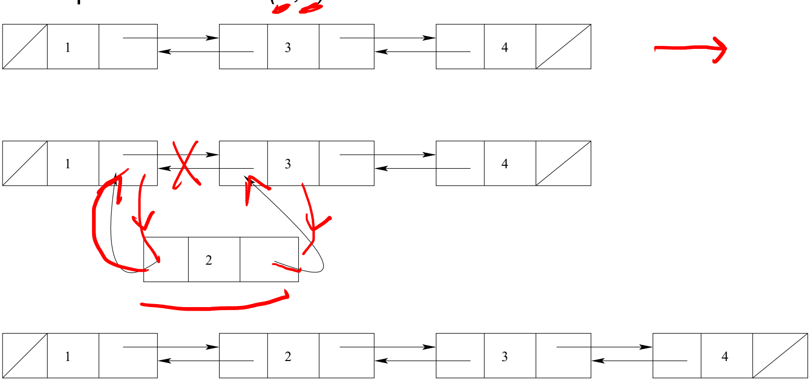
INSERTAFTER(p, e)

```
▷ Create a new node v
v.element ← e
▷ Link v to its predecessor
v.prev ← p
▷ Link v to its successor
v.next ← p.next
▷ Link p's old successor to v
(p.next).prev ← v
▷ Link p to its new successor v
p.next ← v
return v
```



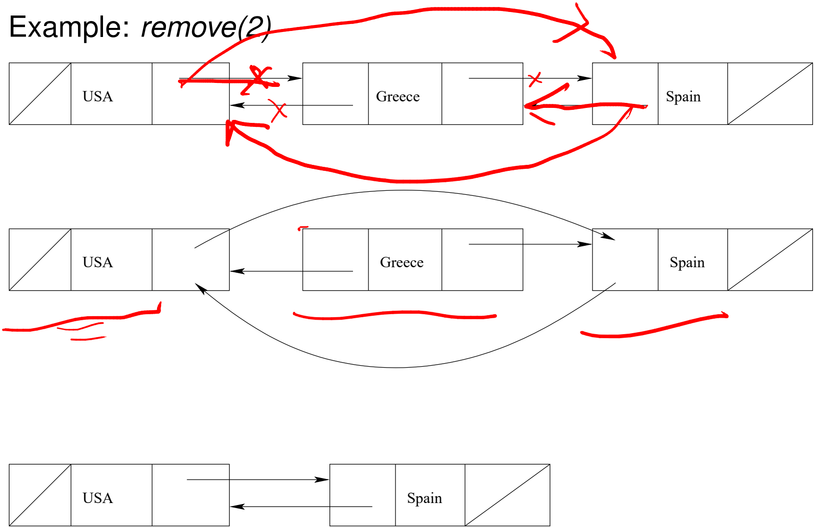
List update: Element insertion

Example: *insertAfter(1,2)*



List update: Element removal

Example: *remove(2)*



List update: Complexity

What is the cost of the *insertion* and *removal* methods?

- If the address of element p is known, then the cost is $O(1)$.
- If only the address of the *head* of the list is known, then the cost of an update is $O(p)$ (we need to traverse the list from positions $0, \ldots, p$ to first find p).

Ordered Data

We often wish to store data that is ordered in some fashion (typically in numeric order, or alphabetical order, but there may be other ways of ordering it).

Ordered Data

We often wish to store data that is ordered in some fashion (typically in numeric order, or alphabetical order, but there may be other ways of ordering it).

Arrays and lists are obvious structures that may be used to store this type of data.

Ordered Data

We often wish to store data that is ordered in some fashion (typically in numeric order, or alphabetical order, but there may be other ways of ordering it).

Arrays and lists are obvious structures that may be used to store this type of data.

Based on our previous discussions, arrays aren't generally efficient to maintain data where we must add/delete items *and* maintain a sorted order. Linked lists may provide a better method in that case, but it is generally harder to *search for* an item in a list.

Binary Search - Algorithm

Here's a recursive search algorithm.

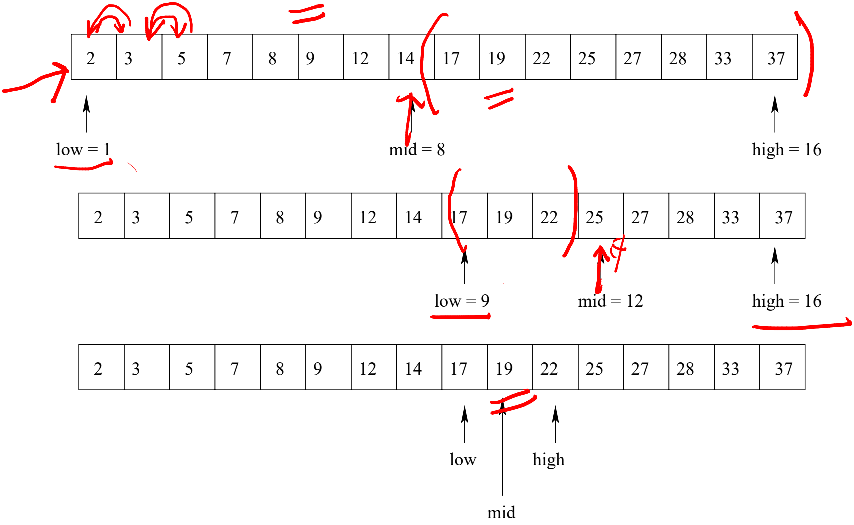
BINARYSEARCH(S, k, low, $high$)

$\triangleright$ Input is an ordered array of elements.
$\triangleright$ Output: Element with key k if it exists, otherwise an error.

if $low > high$
    then return *NO_SUCH_KEY*
else
        $mid \leftarrow \lfloor (low + high)/2 \rfloor$
        if $k = key(mid)$
            then return $elem(mid)$
        elseif $k < key(mid)$
            then return BINARYSEARCH(S, k, low, mid-1)
        else return BINARYSEARCH(S, k, $mid + 1$, $high$)

Binary Search - Example

$\textsc{BinarySearch}(S, 19, 1, \mathit{size}(S))$



Complexity of Binary Search

Let the function $T(n)$ denote the running time of the binary search method.

We can characterize the running time of the recursive binary search algorithm as follows

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ T(n/2) + b & \text{otherwise} \end{cases}$$

where b is a constant that denotes the time for updating *low* and *high* and other overhead.

Complexity of Binary Search

Let the function $T(n)$ denote the running time of the binary search method.

We can characterize the running time of the recursive binary search algorithm as follows

$T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$$T(n) = \begin{cases} \cancel{b}\ c & \text{if } n < 2 \\ T(n/2) + \cancel{b}\ c & \text{otherwise} \end{cases}$$

where b is a constant that denotes the time for updating *low* and *high* and other overhead.

As you did in COMP108, it can be shown that binary search runs in time $O(\log n)$ on a list with n keys.

$n^{\log_b a} = n^0 = 1$

$a = 1, \quad b = 2, \quad \boxed{f(n) = c}$

Complexity of Binary Search (cont.)

Comparison of linked list vs. a lookup table (sorted array).

Method	Linked list	Lookup table
findElement	$O(n)$	$O(\log n)$
insertItem (having located its position)	$O(1)$	$O(n)$
removeElement	$O(n)$	$O(n)$

(WE DON'T KNOW THE LOCATION IN THE LIST)

Can we find something else?

We see some of the contrast between linked lists and arrays.

List are somewhat costly (in terms of time) to find an item, but it is easy (i.e. quick) to add an item once we know where to insert it.

Arrays are efficient when we search for items, but it is expensive to update the list and maintain items in a sorted order.

Can we find something else?

We see some of the contrast between linked lists and arrays.

List are somewhat costly (in terms of time) to find an item, but it is easy (i.e. quick) to add an item once we know where to insert it.

Arrays are efficient when we search for items, but it is expensive to update the list and maintain items in a sorted order.

Can we find a data structure that might let us mimic the efficiency of binary search on arrays ($O(\log n)$), *and* lets us insert/delete items more efficiently than arrays or lists (possibly as big as $O(n)$) to maintain an ordered collection?

Another option

One data structure that seems to have this possibility is a *binary search tree.*

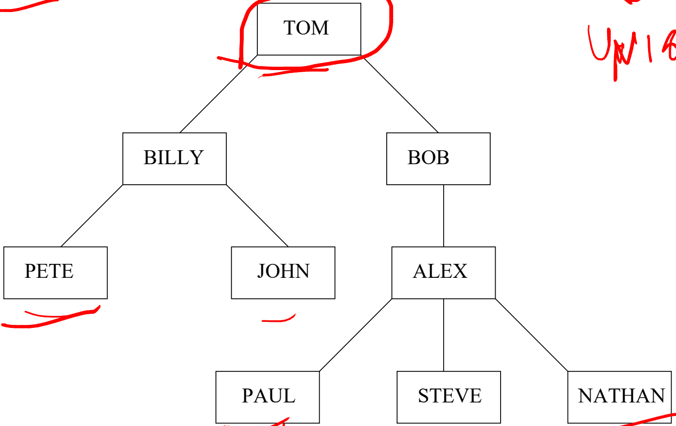
We pause first to review *rooted trees* before we get to binary search trees.

You have previously encountered binary search trees in COMP 108.

Data Structures: Rooted Trees

- A *rooted tree*, T, is a set of nodes which store elements in a parent-child relationship.
- T has a special node, r, called the *root* of T.
- Each node of T (excluding the root node r) has a *parent* node.

UNIQUE

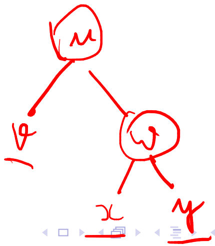


Rooted trees: terminology

- If node u is the *parent* of node v, then v is a *child* of u.
- Two nodes that are *children* of the same *parent* are called *siblings*.
- A node is a *leaf* (external) if it has no *children* and *internal* otherwise.
- A tree is *ordered* if there is a *linear* ordering defined for the *children* of each internal node (i.e. an internal node has a distinguished first child, second child, etc).

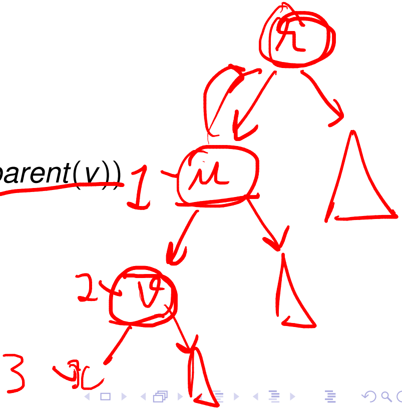
Binary Trees

- A *binary tree* is a rooted ordered tree in which every node has *at most two children*.
- A binary tree is *proper* if each internal node has *exactly two children*.
- Each *child* in a binary tree is labeled as either a *left child* or a *right child*.



- The *depth* of a node, v, is number of ancestors of v, excluding v itself. This is easily computed by a recursive function.

```
DEPTH(T, v)
    if T.isRoot(v)                                 X
        then return 0
        else return 1 + DEPTH(T, T.parent(v))
```



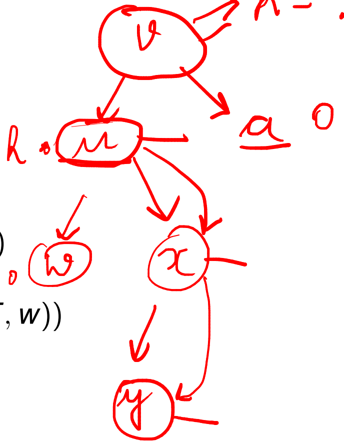
$2 = \underline{1+1}$

The height of a tree

- The *height* of a tree is equal to the maximum depth of an external node in it. The following pseudo-code computes the height of the subtree rooted at v.

$h = 3$

```
HEIGHT(T, v)

if ISEXTERNAL(v)
    then return 0
    else
        h = 0
        for each w ∈ T.CHILDREN(v)
            do
                h = MAX(h, HEIGHT(T, w))
        return 1 + h
```



$1+1+1 = 3$ $h = 0$

Traversal of trees

There are essentially three ways that trees are generally explored/traversed. These methods are the

1. Pre-order traversal
2. Post-order traversal
3. In-order traversal

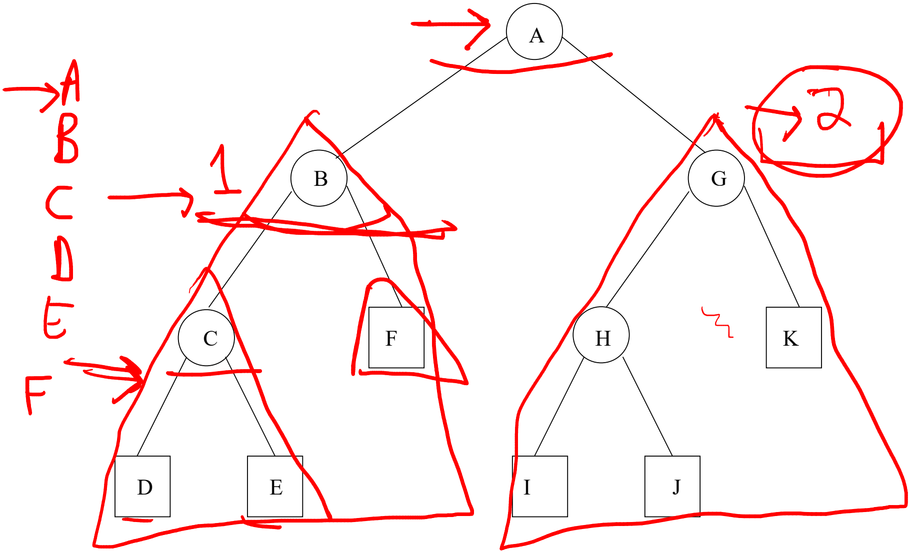
Preorder traversal in trees

- In a *preorder traversal* of a tree, T, the *root* of T is visited first, then the subtrees rooted at its children are traversed recursively.

```
PREORDER(T, v)
    PRINT(v)
    for each child w of v
        do
            PREORDER(T, w)
```

1st LEFT, THEN RIGHT

Preorder traversal in trees (cont.)



- *preorder(T,A)* would produce: A,B,C,D,E,F,G,H,I,J,K.

Postorder traversal of trees

- In a *postorder traversal* of an ordered tree, T, the root of T is visited *last*.

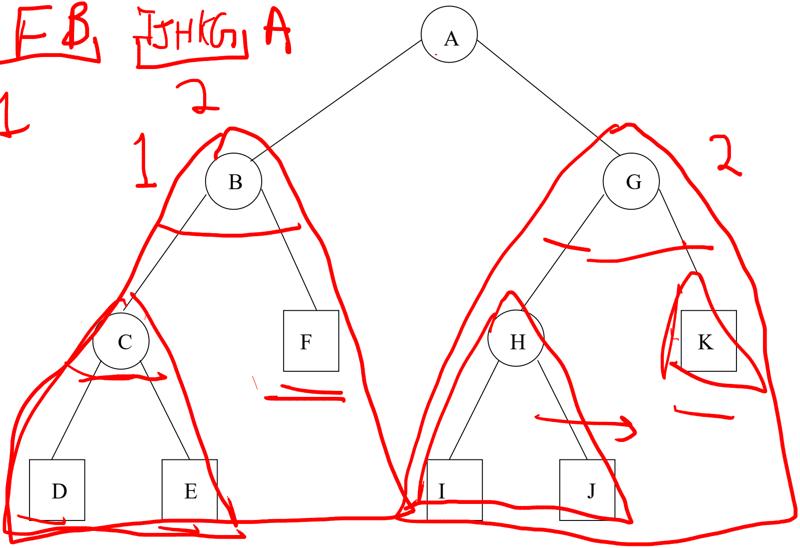
```
POSTORDER(T, v)
for each child w of v
        do
            POSTORDER(T, w)

PRINT(v)
```

→ 1st LEFT, THEN RIGHT

Postorder traversal of trees

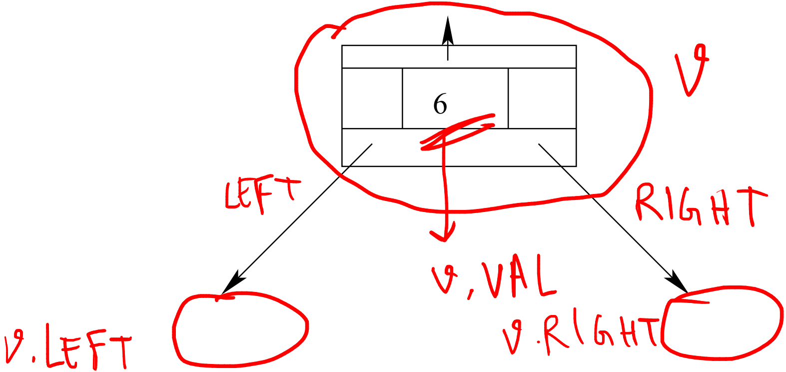
DEC FB IJHKG A

1 2



- Post-order traversal: D,E,C,F,B,I,J,H,K,G,A.

- Linked structure: each node v of T is represented by an *object* with *references* to the *element* stored at v and positions of its parents and children.

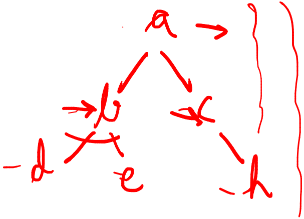


2^h

For rooted trees where each node has at most t children, and is of bounded depth, you can store the tree in an array A. This is most useful when you're working with (almost) complete t-ary trees.

Consider, for example, a *binary* tree. The root is stored in $A[0]$.

The (possibly) two children of the root are stored in $A[1]$ and $A[2]$. The two children of $A[1]$ are stored in $A[3]$ and $A[4]$, and the two children of $A[2]$ are in $A[5]$ and $A[6]$, and so forth.



$A = [a, b, c, d, e, \bot, h]$

0 1 2

Data structures for rooted t-ary trees (cont.)

In general, the two children of node $A[i]$ are in $A[2 * i + 1]$ and $A[2 * i + 2]$.

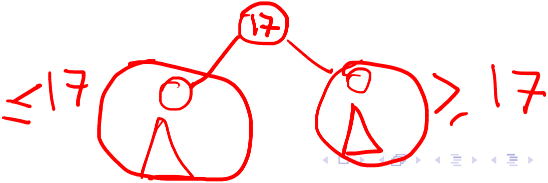
The parent of node $A[i]$ (except the root) is the node in position $A[\lfloor (i - 1)/2 \rfloor]$.

This can be generalized to the case when each node has at most t children.

A Binary Search Tree (*BST*) applies the motivation of binary search to a *tree-based* data structure.

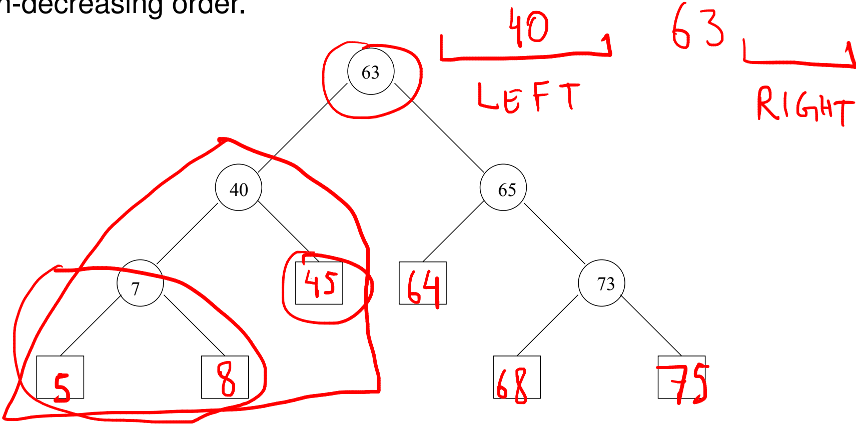
In a *BST* each internal node stores an element, *e* (or, more generally, a key *k* which defines the ordering, and some element *e*).

A *BST* has the property that, given a node *v* storing the element *e*, all elements in the *left subtree* at *v* are *less than or equal to e*, and those in the *right subtree* at *v* are *greater than or equal to e*.



LEFT, r, RIGHT

An *inorder* traversal of a *BST* visits the nodes in non-decreasing order.



Searching in a BST

Here's a recursive searching method:

$k = 71$

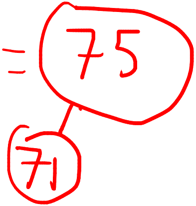
TREESEARCH(k, v)

▷ Input: A key k and a node v of a binary search tree.

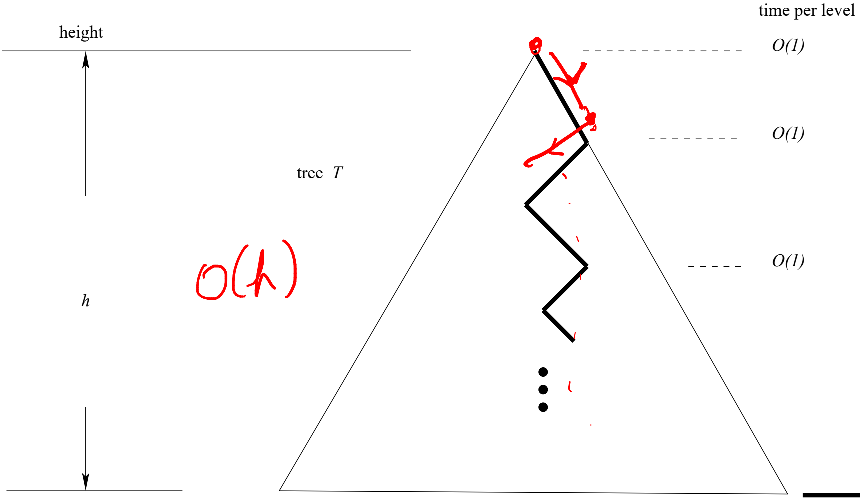
▷ Output: A node w in the subtree $T(v)$, either w is an internal node with key k or w is an external node where the key k would belong if it existed.

```
if ISEXTERNAL(v)
    then return v
if k = key(v)
    then return v
elseif k < key(v)
    then return TREESEARCH(k, T.LEFTCHILD(v))
else return TREESEARCH(k, T.RIGHTCHILD(v))
```

$v =$

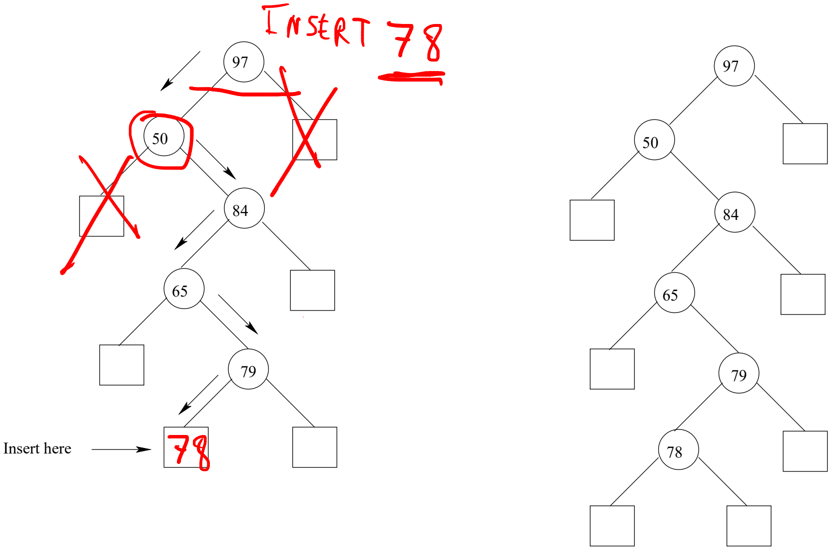


Complexity of searching in a BST



total time: $O(h)$

Insertion in a BST

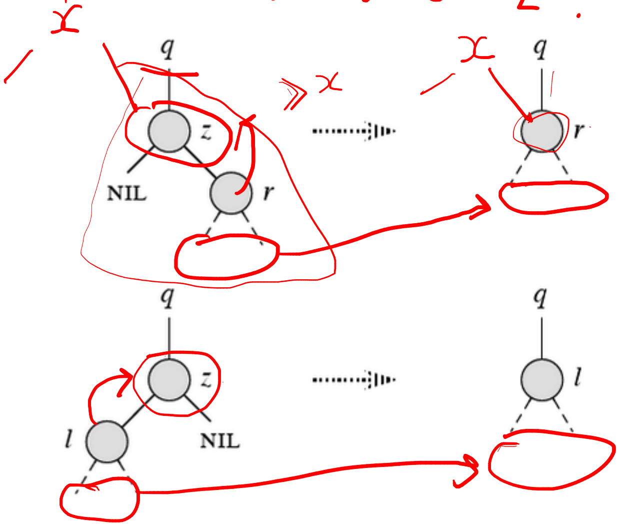


Insertion of 78 is performed in time $O(h)$.

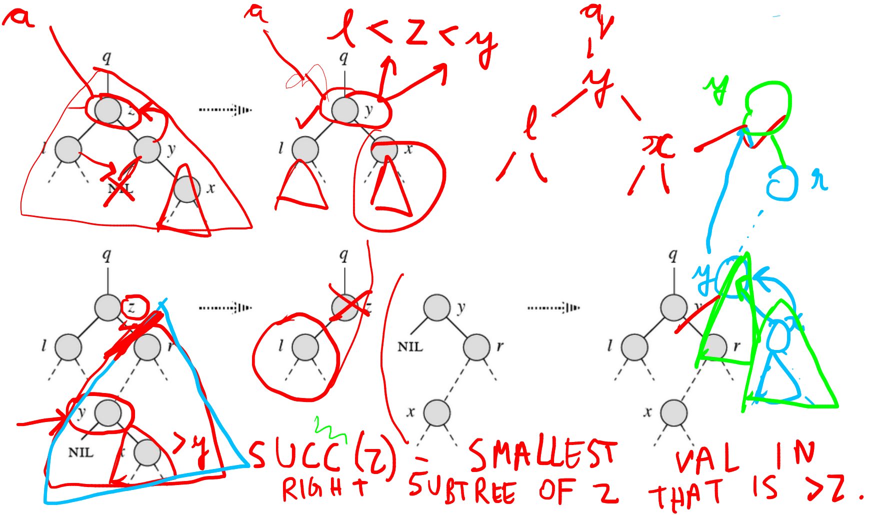
Deletion in a BST

Several cases!

DELETE z.



Deletion in a BST (cont.)



In general, deletion takes $O(h)$ time.

Inefficiency of general BSTs

- All operations in a BST are performed in $O(h)$, where h is the height of the tree.

  Unfortunately, h may be as large as n, e.g. for a *degenerate tree*.

- The main advantage of binary searching (i.e. the $O(\log n)$ search time) may be lost if the BST is not *balanced*. In the worst case, the search time may be $O(n)$.

- There are variants of BSTs that are *self-balancing*. e.g., AVL trees, red-black trees, but we won't cover these in this module.

Priority Queues

A *Priority Queue* is a container of elements, each having an associated *key*.

Keys determine the *priority* used in picking elements to be removed.

A *priority Queue* (PQ) has these fundamental methods:

- *insertItem(k,e)*: insert element *e* having key *k* into PQ.
- *removeMax()*: remove maximum element. (MAX KEY)
- *maxElement()*: return maximum element.
- *maxKey()*: return key of maximum element.

(Of course, we could have priority queues that are based on maintaining the minimum elements.)

Priority Queue - Sorting

How can we use a priority queue to perform sorting on a set C? Do this in two phases:

- *First phase*: *Put* elements of C into an initially empty priority queue, P, by a series of n *insertItem* operations.
- *Second phase*: *Extract* the elements from P in *non-increasing* order using a series of n *removeMax* operations.

PQ Sorting - Algorithm

PQ-SORT(C, P)

$\triangleright$ Input: An n element sequence C and a priority queue P.

$\triangleright$ Output: The sequence C sorted using the total order relation.

while $C \neq \emptyset$

do

$e \leftarrow$ C.REMOVEFIRST() → REMOVE '1st' ELEM OF DATASET

P.INSERTITEM(e, e) → ADD TO PQ

while $P \neq \emptyset$

do

$e \leftarrow$ P.REMOVEMAX()

C.INSERTLAST(e)

Heap Data Structure

INSERT ITEM
REMOVE MAX
$O(\log n)$

A *heap* is a realization of a Priority Queue that is *efficient* for both *insertions* and *deletions*.

A *heap* allows insertions and deletions to be performed in *logarithmic* time.

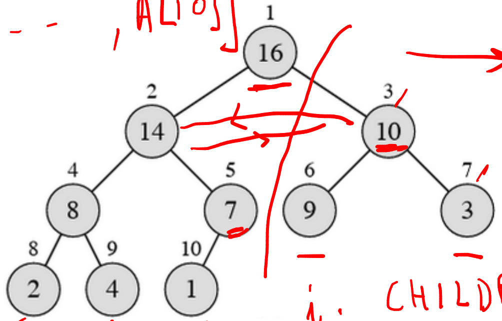
In a *heap* the *elements* and their *keys* are stored in an almost complete binary tree. Every level of the binary tree, except possibly the last one, will have the maximum number of children possible.

Heap-order property

ROOT : MAX

- In a *heap* T, for every *node* v (excluding the *root*) the *key* at v is *less* than (or *equal* to) the *key* stored at its parent.

A: [A[1] - - - , A[10]]



> 16 IN BST

i : CHILDREN ARE $2i$, $2i+1$

PARENT : $\left\lfloor \frac{i}{2} \right\rfloor$

PQ/Heap implementation

An efficient realization of a heap can be achieved using an array for storing the elements (i.e. the vector representation of a tree that we discussed earlier).

So a heap can be implemented with the following:

- *heap*: A (nearly complete) binary tree T containing elements with *keys* satisfying the *heap-order property*, stored in an array.
- *last*: A *reference* to the *last used* node of T in this array representation.
- *comp*: A *comparator* function that defines the *total order relation* on *keys* and which is used to maintain the *maximum* (or minimum) element at the *root* of T.

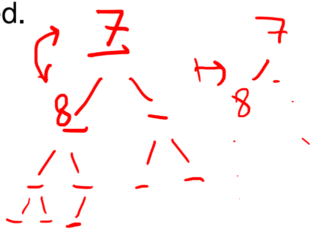
10.

7 8

HEAP ≡ MAX-HEAP

Inserting a new item into a heap begins by adding this element to the bottom of the tree in the position of the first unused, or empty, child.

Then, if necessary, this new element “bubbles” its way up the heap until the heap-order property is restored.



Deletion in a heap

Deletion in a heap consists of removing the minimum (or maximum) element (at the root) from the heap. Then the bottom, right-most element in the heap (the element at the end of the array that stores the heap) is moved to the root.

To restore the heap-order property, this item then “sinks” or bubbles down the heap.

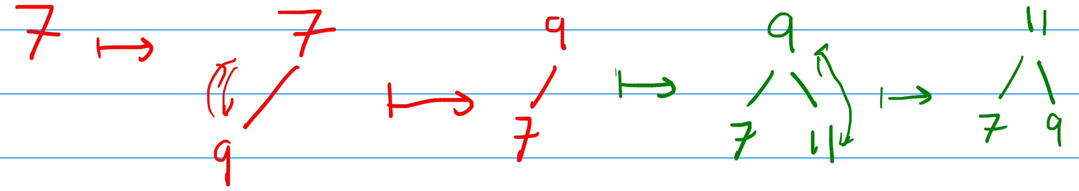
Deletion in a heap

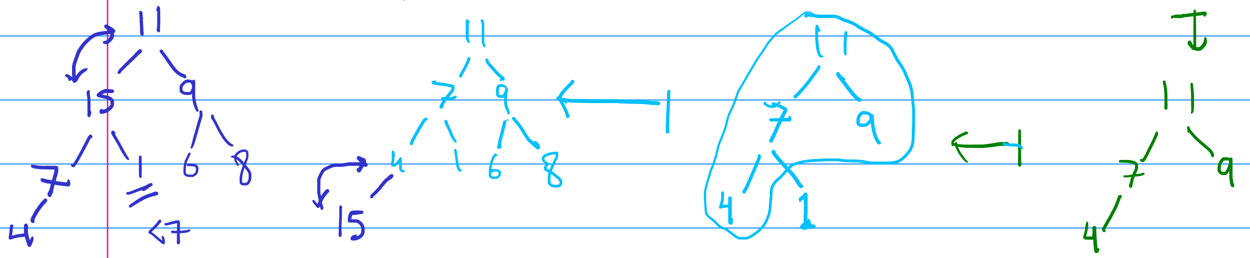
Deletion in a heap consists of removing the minimum (or maximum) element (at the root) from the heap. Then the bottom, right-most element in the heap (the element at the end of the array that stores the heap) is moved to the root.

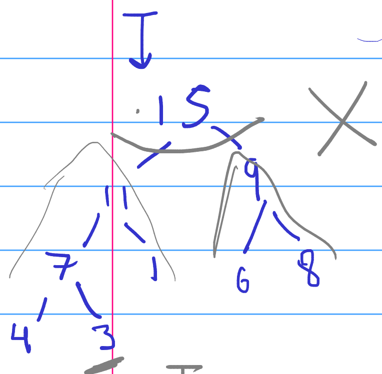
To restore the heap-order property, this item then "sinks" or bubbles down the heap.

An example of both of these operations will be (was) shown in the lecture.

~~7~~, ~~9~~, ~~11~~, ~~4~~, ~~1~~, ~~6~~, ~~8~~, ~~15~~, 3







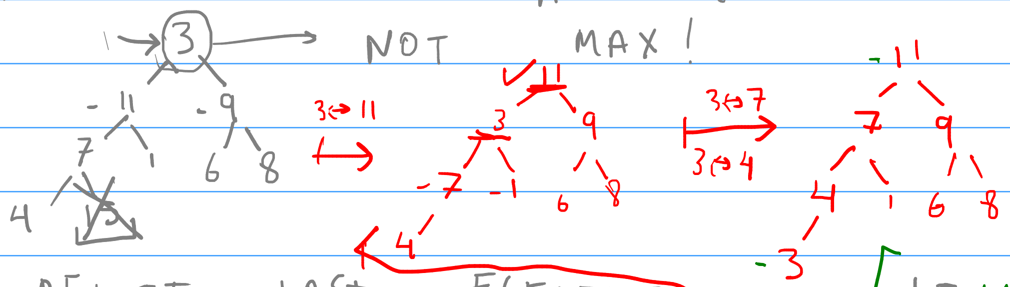
$\boxed{n \text{ NODES} \Rightarrow \text{DEPTH} : O(\log n)}$

EVERY INSERTION CAUSES $O(\log n)$ SWAPS.

BUILD HEAP: $O(n \log n)$.
CAN DO IN $O(n)$ TIME!

$O(1)$ → 1) SWAP ROOT WITH LAST ELEMENT

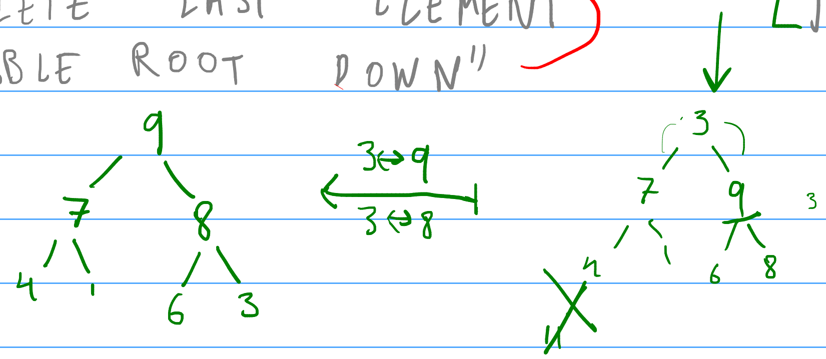
3 → NOT MAX!



[15, 11, 9, 8, 7, 6, 4, 3, 1]

$O(1)$ → 2) DELETE LAST ELEMENT

$O(\log N)$ → 3) "BUBBLE ROOT DOWN"



Heap performance

Since a heap is an almost-complete binary tree, it stores n items in a tree of height $O(\log n)$. Thus we have the following summary of running times for operations that can be performed on a heap.

Operation	**time**
size, isEmpty	$O(1)$
maxElement, maxKey	$O(1)$
insertItem	$O(\log n)$
removeMax	$O(\log n)$

Heap-Sorting

Heap-sort: “Convert” input array into a heap. Perform n removeMax operations.

Theorem: The heap-sort algorithm sorts a sequence, S, of n comparable items in $O(n \log n)$ time, where

- Bottom-up construction of heap with n elements takes $O(n \log n)$ time, and
- Extraction of n elements (in de~~in~~creasing order) from the heap takes $O(n \log n)$ time.