Array ADT: Example

- Represent a n-dimensional vector by an array of length n. Implement the dimension-product given two arrays a_1 and a_2 and output to a_3 such that $a_3[i] = a_1[i] \cdot a_2[i]$

```
ADT Array
 Array createArray(n)
 Item retrieve(arr, i)
 Item store(arr, i, itemToStore)
```

Algorithm 2: ***Dimension Product***
Input: Vector a_1, a_2, a_3 of length n
Output: Dimension-product of a_1 and a_2 which stored in a_3.

```
int i,i_dimension_coordinate;
for (i=0; i<n; i++){
     i_dimension_coordinate = retrieve(a1,i)*retrieve(a2,i);
     store(a3, i, i_dimension_coordinate);
}
return a3;
```

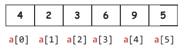
Implementing Binary Search with Recursion

Algorithm 2: ***BinarySearch(arr, searchnum, left, right)***
Input: An sorted array arr, an integer searchnum,
the left position of the active range, the right position of the active range
Output: the index i such that arr[i] = searchnum, or -1 if no such i exists.

```
if( left == right ) // base case
   return left if arr[left] == searchnum otherwise -1;
int middle = (left + right)/2;
if (arr[middle] == searchnum)
     return middle;
else if (arr[middle] < searchnum)
     return BinarySearch(arr, searchnum, middle+1, right);
else
     return BinarySearch(arr, searchnum, left, middle -1);
```

Practice

- Finding the maximum number from an array of integers.
  - What is the recursion?
    - Let f(n) be the maximum number of the first n elements in the array
    - f(6) = max(5, f(5));
    - f(5) = max(9, f(4));
    - f(4) = max(6, f(3));
    - …
    - f(1) = 4;

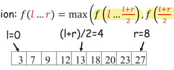


Algorithm P1: ***recursiveMax1(arr,n)***

```
if n==1
  return arr[0]
return max(arr[n-1], recursiveMax1(arr,n-1))
```

Practice (Cont.)

- Another recursive approach:
  - Let $f(l \ldots r)$ be the maximum of the l-th element to the r-th element in the array arr.
  - Base case: $f(l \ldots l) = arr[l]$
  - Recursion: $f(l \ldots r) = \max\left(f\left(l \ldots \frac{l+r}{2}\right), f\left(\frac{l+r}{2} + 1 \ldots r\right)\right)$



Algorithm P2: ***recursiveMax2(arr,left,right)***

```
if left==right
  return arr[left]
return max(recursiveMax2(arr, left, (left+right)/2),
           recursiveMax2(arr, (left+right)/2+1,right))
```

Worst Case Analysis: BinarySearch

- How many times do we need to execute while loops?
  - Each time we reduce the size by half
  - The first time, n, the second time, $n/2$, the i-th time: $\frac{n}{2^{i-1}}$
  - In the last iteration, we have only one element when $\frac{n}{2^{i-1}} = 1$

IterativeBinarySearch*(A, n,searchnum)

```
left = 0, right = n-1                2
while left <= right                  log2(n) + 1
    middle = (left+right)/2          3·log2(n) + 3
    if A[middle] < searchnum         2·log2(n) + 2
       left = middle +1              2·log2(n) + 2
    elseif A[middle] > searchnum     2·log2(n) + 2
       right = middle -1             2·log2(n) + 2
    else                             1
       return middle                 1
return -1                            1
```

*We use iterative to indicate that the algorithm is implemented with loops

Total: $12 \cdot \log_2(n) + 17$

Practice: Using Sum Property

- What is the time complexity (using Big-Oh) of maxSubArraySum?

maxSubArraySum(A, n)

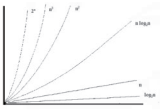
```
maxSum=A[0]
for i=0 to n-1
  subSumFromI=0
  for j = i to n-1
    subSumFromI+=A[j]
    if subSumFromI>maxSum
      maxSum=subSumFromI
return maxSum
```

nested

$g_1(n) = O(1)$
$g_2(n) = O(n)$
$g_3(n) = O(n)$
$g_4(n) = \boxed{O(n^2)}$
$g_5(n) = \boxed{O(n^2)}$
$g_6(n) = \boxed{O(n^2)}$
$g_7(n) = \boxed{O(n^2)}$
$g_8(n) = O(1)$

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

Common Big-Oh Functions/Classes



	n=1	n=2	n=4	n=8	n=16	n=32
(1)	1	1	1	1	1	1
$(\log_2 n)$	0	1	2	3	4	5
(n)	1	2	4	8	16	32
$(n \log_2 n)$	0	2	8	24	64	160
(n^2)	1	4	16	64	256	1024
(n^3)	1	8	64	512	4096	32768
(2^n)	2	4	16	235	65536	4294967296

BETTER ↕ WORSE

$O(1)$	constant time
$O(\log n)$	log time
$O(n)$	linear time
$O(n \cdot \log n)$	log linear time
$O(n^2)$	quadratic time
$O(n^3)$	cubic time
$O(2^n)$	exponential time
O(n!)	

Practice

- Prove the sum property
  - $g_1(n) = \Omega(f_1(n)), g_2(n) = \Omega(f_2(n))$
    - Then, $g_1(n) + g_2(n) = \Omega(\max\{f_1(n), f_2(n)\})$

Proof: According to the definition, we know that there exist constants c_1, n_1 such that

$$g_1(n) \geq c_1 \cdot f_1(n) \text{ for } n \geq n_1,$$

And there exist constants c_2, n_2 such that

$$g_2(n) \geq c_2 \cdot f_2(n) \text{ for } n \geq n_2$$

Then, for $n \geq n_1$and $n \geq n_2$, i.e., $n \geq \max(n_1, n_2)$ we know that

$$g_1(n) + g_2(n) \geq \min(c_1, c_2) \max\{f_1(n), f_2(n)\}.$$

According to the definition, we know $g_1(n) + g_2(n) = \Omega(\max\{f_1(n), f_2(n)\})$

Counting Basic Operations in Recursion

- Given input size n, let $g(n)$ be the total # of basic operations executed in BinarySearch in the worst case.

BinarySearch(arr, searchnum, left, right)

```
1  if left == right
2     if arr[left]= searchnum
3        return left
4     else
5        return -1
6  middle =(left + right)/2
7  if arr[middle] = searchnum
8     return middle
9  elseif arr[middle] < searchnum
10    return BinarySearch(arr, searchnum, middle+1, right)
11 else
12    return BinarySearch(arr, searchnum, left, middle -1)
```

We can still count the number of basic operations for this part

The total number of basic operations executed is a constant independent of the input size n, we can use a to denote this

We either run line 10 or 12, but not both. What is the number of basic operations that are executed by Line 10 or 12?

Analysis for Recursive Binary Search (ii)

- Given $g(n) = a + g\left(\frac{n}{2}\right), g(1) = b$
  - What is $g(4)$ by using a and b to represent?

$$\begin{aligned} g(4) &= g(2) + a \\ &= g(1) + a + a \\ &= 2a + b \end{aligned}$$

  - What is $g(n)$ by using a and b to represent if $n = 2^x$?

$$\begin{aligned} g(n) &= g\left(\frac{n}{2}\right) + a \\ &= g\left(\frac{n}{2^2}\right) + a + a \\ &= g\left(\frac{n}{2^3}\right) + a + a + a \\ &= g\left(\frac{n}{2^4}\right) + a + a + a + a \\ &= \cdots \\ &= g(1) + \overbrace{a + a + \cdots + a + a}^{x \text{ of them}} = x \cdot a + b = a \cdot \log_2 n + b \end{aligned}$$

Analysis for Selection Sort (i)

- Let $g(n)$ be the total number of basic operations in the worst case. Let $g(1) = b$
  - We have that:
    - $g(n) = g(n-1) + O(n)$
  - We can find constant c such that
    - $g(n) \leq g(n-1) + c \cdot n$
- We have that:
  - $g(n) \leq g(n-1) + c \cdot n \leq g(n-2) + c \cdot n + c \cdot (n-1)$
    $\leq g(n-3) + c \cdot n + c \cdot (n-1) + c \cdot (n-2)$
    $\leq g(1) + c \cdot n + c \cdot (n-1) \cdots + c \cdot 2$
    $\leq c \cdot \frac{n(n+1)}{2} + b$
  - $g(n) = O(n^2)$

In many cases, we only want to have the upper bound the worst case running time. Deriving its Big-Oh is sufficient.

Master Theorem (Big-Oh Version)

- Let $g(n)$ be the running cost depending on the input size n, and we have its recurrence:
  - $g(1) = O(1)$
  - $g(n) \leq a \cdot g\left(\left\lceil\frac{n}{b}\right\rceil\right) + O(n^\lambda)$
  - With a, b, λ to be constants such that $a \geq 1, b > 1, \lambda \geq 0$. Then,
    - If $\log_b a < \lambda$, $g(n) = O(n^\lambda)$
    - If $\log_b a = \lambda$, $g(n) = O(n^\lambda \cdot \log n)$
    - If $\log_b a > \lambda$, $g(n) = O(n^{\log_b a})$.
  - Limitations: cannot be applied to cases like:
    - $g(n) \leq a \cdot g(n-1) + c$

Linked List: Initialization

- Given a linked List L.
  - What L includes: $L.head$, $L.tail$, which stores the head and tail of the linked list.
  - For a node p, we use $p.element$ to access the element stored in this node and $p.prev/p.next$ to access the node preceding/following node.
    - The C code will be different, e.g., you may use $p \rightarrow prev$ or $p \rightarrow next$. You will know how to do it in the tutorial next week.
  - For an empty list, simply let the next pointer of head node points to the tail and the prev pointer of tail points to the head.

Algorithm: ***createList()***

```
Allocate memory for list L
Allocate memory for node L.head and node L.tail
L.head.prev = NULL
L.head.next = L.tail
L.tail.prev = L.head
L.tail.next = NULL
```

Searching in Linked List

- Assume that we have the linked list L
  - We can use L.head.next to obtain the first node storing element
  - We can use the next pointer of the first node to access the second node storing element, etc.
  - Thus, we can repeatedly access all the elements in an iterative manner until we reach the tail.
- Example: search on the list

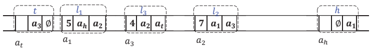
Algorithm: ***ListSearch(L, searchnum)***

```
node = L.head.next
while node != L.tail
   if node.element == searchnum
      return node
   node = node.next
return NULL
```

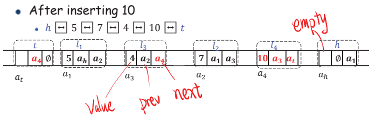
Search cost: $O(n)$

Insertion: View in Memory

- Before the insertion
  - h ⟷ 5 ⟷ 7 ⟷ 4 ⟷ t



- After inserting 10
  - h ⟷ 5 ⟷ 7 ⟷ 4 ⟷ 10 ⟷ t



Insertion: pseudo-code

- Step 1: identify the last node u_{last} stored in the linked list
- Step 2: allocate a new node u_{new}
- Step 3: set $u_{new}.element = e$
- Step 4: set $u_{new}.prev$ to the address of u_{last}
- Step 5: set $u_{new}.next$ to the address of the tail.
- Step 6: set $u_{last}.next$ to the address of u_{new}
- Step 7: set $tail.prev$ to the address of u_{new}

Algorithm: ***insert(L,e)***

```
ulast = L.tail.prev
unew <- allocate a new node from memory
unew.element = e
unew.prev = ulast
unew.next = L.tail
ulast.next = unew
L.tail.prev = unew
```

Insertion in Place: Pseudo-code

- Step 1: allocate a new node u_{new}
- Step 2: set $u_{new}.element = e$
- Step 3: set $u_{new}.prev$ to the address of p
- Step 4: set $u_{new}.next$ to the address of $q = p.next$
- Step 5: set $p.next$ to the address of u_{new}
- Step 6: set $q.\ prev$ to the address of u_{new}

Algorithm: ***insertInPlace(L,p, e)***

```
unew <- allocate a new node from memory
unew.element = e
unew.prev = p
q=p.next; unew.next =q
p.next = unew
q.Prev = unew
```

Deletion: Pseudo-code

- Step 1: identify the preceding node u_{prec} of node p
- Step 2: identify the succeeding node u_{succ} of node p
- Step 3: let $u_{prec}.next$ points to u_{succ}
- Step 4: let $u_{succ}.prev$ points to u_{prec}
- Step 5: free the memory of node p.

Algorithm: ***delete(L,p)***

```
uprec = p.prev
usucc = p.next
Uprec.next = usucc
Usucc.prev = uprec
Free the memory of node p
```

Balanced Symbol Checking Algorithm

- Step 1: make an empty stack
- Step 2: read the symbols from the input text
  - If the symbol is an opening symbol, push it onto the stack
  - If it is a closing symbol
    - If the stack is empty: return false
    - Otherwise, pop from the stack. If the symbol popped does not match the closing symbol, return false
- Step 3: at the end, if the stack is not empty, return false (unbalanced), else return true (balanced)

How to Derive the Postfix?

- 7/(2+3)*4
  - According to the definition, operator should appear after operand. 7/(2+3) and 4 should be put before *
  - Thus, the postfix L for the expression is:
    - L: "Postfix for 7/(2+3)" 4 *
  - We got a smaller problem. What is the postfix L' for 7/(2+3)
    - 7 and postfix of (2+3) should appear before /
    - L': 7 "postfix for (2+3)" /.
    - What is the postfix L" for (2+3)
      - 2 3 +.
  - => Postfix for L' is: 7 2 3 + /
  - => Postfix for L is: 7 2 3 + / 4 *

Postfix Evaluation Algorithm

- Here, we only consider evaluation of expressions using the four binary operators +, -, * and /.
  - Create a stack
  - Scan the postfix expression from left-to-right.
    - If an operand is encountered, push to the stack
    - If an operator is encountered
      - pop the stack for the right hand operand
      - pop the stack for the left hand operand
      - apply the operator to the two operands
      - push the result onto the stack
  - When the postfix expression has been scanned, the result is kept on the top of the stack.

Queue Design with Linked List

- createQueue: create a linked list L

FIFO

Algorithm: ***createQueue(capacity)***

```
Q<-Allocate new memory for queue
Q.L <- allocate new memory for list
return Q
```

- Enqueue: add e to the end of the linked list

Algorithm: ***Enqueue(Q,e)***

```
insert(Q.L,e)
```

- Dequene: retrieve the first element, i.e., $L.head.next.element$, and delete the first node, i.e., $L.head.next$.

Algorithm: ***Dequeue(Q)***

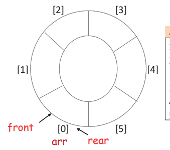
```
if isEmpty(Q.L) error "Queue empty"
frontNode = Q.L.head.next
frontElement = frontNode.element.
delete(Q.L, frontNode)
return frontElement
```

Queue: Array Design (Cont.)

- Enqueue: change the rear position to the next position.
- Dequene: change the front position to the next position.
- In the circular view,
  - The only difference: The next of capacity-1 is 0.
    - If we have capacity =6
      - The next of 0 is 1, the next of 1 is 2,… the next of 4 is 5
      - The next of 5 is 0.
    - The next rear:
      - if rear!= capacity-1 rear ← rear+1, else rear ← 0
      - Or (rear+1)% capacity, where % is the module function that returns the remainder of the division
    - The next front? (front+1)%capacity

Queue ADT Operations: Array Design

- createQueue(capacity): create a queue with a maximum number of capacity elements
  - We allocate an array of size to be capacity,
  - Initialize front = 0 and rear = 0.



Algorithm: ***createQueue(capacity)***

```
Q<- Allocate memory for queue
Q.arr <- allocate an array of
size capacity
Q.front = 0
Q.rear = 0
Q.capacity = capacity
```

Enqueue& Dequeue Pseudocode

- Assume that Q is the queue
  - $Q.arr$ stores the elements
  - $Q.capacity$ stores its maximum size
  - $Q.front$ stores the front position
  - $Q.rear$ stores the rear position

Algorithm: ***enqueue(Q, e)***

```
if isFull(Q)
   error "Queue full"
Q.arr[Q.rear] = e
Q.rear = (Q.rear
         +1)%Q.capacity
```

Algorithm: ***dequeue(Q)***

```
if isEmpty(Q)
   error "Queue empty"
e = Q.arr[Q.front]
Q.front = (Q.front
         +1)%Q.capacity
return e
```

Queue ADT Operations: Array Design

- IsEmpty:

Algorithm: ***isEmpty(Q)***

```
return Q.front ==Q.rear
```

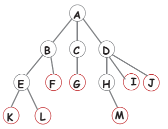
- IsFull: The maximum number of elements we can store with the array is: Q.capacity
  - When we store Q.capacity elements, what is the relationship between Q.front and Q.rear?
    - Also Q.front = Q.rear!
    - Cannot distinguish if the queue is empty or full
  - How to handle this issue?
    - we store at most Q.capacity-1 elements

Algorithm: ***isFull(Q)***

```
return Q.front ==(Q.rear+1)%Q.capacity
```

Some Terminology in Trees (Cont.)

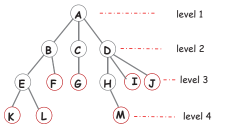
- Children: the roots of the subtrees of a node X are the children of X
- Parent: X is the parent of its children
- Siblings: children of the same parent are said to be siblings.
- Ancestors of a node: parent, grand-parent, grand-grandparent,etc.
- Descendants of a node: child, grandchild, grand grand-child, etc.



- E and F are the children of B
- B is the parent of E and F
- C is the sibling of B
- A, B, E are the ancestors of K
- H, I, J, M are the descendants of node D

Some Terminology in Trees (Cont.)

- The level of a node: defined by letting the root be at level one. If a node is at level l, then it children are at level l+1.
- Height (or depth): the maximum level of any node in the tree

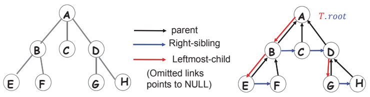


# of Nodes
=
# of Edges + 1

The height (depth) of the tree is 4

Representation Of Trees

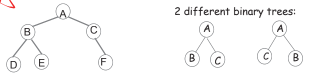
- Pointer representation of a tree T
  - Maintain $root$ to store the address of the root node.
  - For each node, it further maintains the address of its parent, leftmost-child, and right-sibling.
    - For example, for node B
      - B.parent stores the address of node A.
      - B.right-sibling stores the address of node C.
      - B.leftmost-child stores the address of node E.



Binary Trees

- A special tree:
  - Any node in the tree can have at most two children
- Definition
  - A *binary tree* is a finite set of nodes that is
    - either empty or
    - consists of a root and two disjoint binary trees called the left subtree and the right subtree
  - left subtree and the right subtree are distinguished

2 different binary trees:

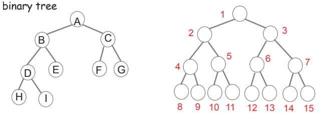


Binary Trees (Cont.)

- The maximum number of nodes on level i of a binary tree is 2^{i-1} for $i \geq 1$.
  - Level 1: 1 node; level 2: 2 nodes; level 3: 4 nodes, etc.
- The maximum number of nodes in a binary tree of height k is $2^k - 1$ for $k \geq 0$.
  - Max # of nodes (level 1) + max # of node (level 2) + … + max # of node (level k)
    $= \sum_{i=1}^{k} 2^{i-1} = 2^k - 1$.
- The height of a binary tree with n nodes falls in the range between $\lceil \log_2(n+1) \rceil$ and n.
  - Why?

Complete Binary Tree (Cont.)

- Given a full binary tree of size $2^h - 1$ of height h, with IDs increasing from top to down, left to right
  - A complete binary tree of size n $(2^{h-1} \leq n \leq 2^h - 1)$ of height h is a binary tree that discards the nodes of the full binary tree with IDs from $n+1$ to $2^h - 1$
- In the following example, we have 9 nodes. The complete binary tree should corresponds to the nodes numbered 1-9 in the full binary tree



Binary Tree: Pointer Implementation

- Create(bintree)

Algorithm: *create(bintree)*
```
bintree = NULL
```

- isEmpty(bintree)

Algorithm: *isEmpty(bintree)*
```
return bintree == NULL
```

- MakeBT(bintree1,element,bintree2)

Algorithm: *MakeBT(bintree1, element, bintree2)*
```
rootNode<- allocate new memory
rootNode.element = element
rootNode.parent = NULL
rootNode.leftchild = bintree1
rootNode.rightchild = bintree2
if bintree1 != NULL
  bintree1.parent = rootNode
if bintree2 != NULL
  bintree2.parent = rootNode
return  rootNode
```

Binary Tree: Pointer Implementation

- Lchild(bintree)

Algorithm: *Lchild(bintree)*
```
if bintree == NULL
  error “empty tree”
return bintree.leftchild
```

- Rchild(bintree)

Algorithm: *Lchild(bintree)*
```
if bintree == NULL
  error “empty tree”
return bintree.rightchild
```

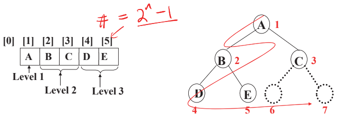
- Data(bintree)

Algorithm: *Data(bintree)*
```
if bintree == NULL
  error “empty tree”
return bintree.element
```

Binary Tree Design (ii)

- An array representation
  - Given a complete binary tree with n nodes. For any i-th node, $1 \leq i \leq n$,
    - parent(i) is $\lfloor i/2 \rfloor$
    - leftChild(i) is at $2i$ if $2i \leq n$. Otherwise, i has no left child.
    - rightChild(i) is at $2i+1$ if $2i+1 \leq n$. Otherwise, i has no right child.

$\# = 2^n - 1$

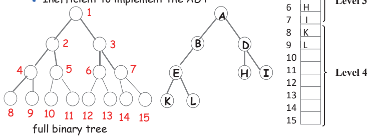


Binary Tree Design (ii)

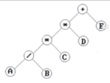
Create arr.size = 2^n
∵ arr[0] is empty

- An array representation
  - Generalize to all binary trees
  - Efficient for complete binary trees.
  - But inefficient for skewed binary trees.
  - Inefficient to implement the ADT

$2^n - 1 = 15$
$n = 4$



Use of Binary Tree ADT



- Binary Tree Traversals
  - Inorder traversal
  - Postorder traversal
  - Preorder traversal
- Arithmetic Expression using binary tree
  Example: A / B * C * D + E
  - Inorder traversal (infix expression)
    A / B * C * D + E
  - Postorder traversal (postfix expression)
    A B / C * D * E +
  - Preorder traversal (prefix expression)
    + * * / A B C D E

L D R
L R D
D L R

Inorder traversal

- Inorder traversal of a binary tree works as follows:
  - Inorder traverse its left-subtree;
  - Then visit the root node;
  - Finally inorder traverse the right-subtree.

Algorithm: *Inorder(btree)*
```
if !isEmpty(btree)
  Inorder(Lchild(btree))
  print Data(btree)
  Inorder(Rchild(btree))
```

L
D
R

```
ADT BinTree
 BinTree create(bt);
 Boolean isEmpty(bt)
 BinTree MakeBT(bt1,
item, bt2);
 BinTree Lchild(bt);
 BInTree Rchild(bt);
 element Data(bt);
```

- What is the output if we feed a binary tree to Algorithm ***Inorder***?

Use of Binary Tree ADT: More Examples

- Calculate the height of a binary tree.

Algorithm: *height(btree)*
```
if isEmpty(btree)
  return 0
lheight ← height(Lchild(btree))
rheight ← height(Rchild(btree))
return 1+ max(lheight, rheight)
```

```
ADT BinTree
 BinTree create(bt);
 Boolean isEmpty(bt)
 BinTree MakeBT(bt1,
item, bt2);
 BinTree Lchild(bt);
 BInTree Rchild(bt);
 element Data(bt);
```

- Sum over all the elements in the binary tree

Algorithm: *sum(btree)*
```
if isEmpty(btree)
  return 0
lsum ← sum(Lchild(btree))
rsum ← sum(Rchild(btree))
return Data(btree) + lsum+ rsum
```

Practice

- Design an algorithm to search from the binary tree and return the node that stores the element if the binary tree contains such an element. Otherwise, return NULL.

Algorithm: *search(btree, searchnum)*
```
if isEmpty(btree)
  return NULL
if Data(btree) == searchnum
  return btree
ls = search(Lchild(btree), searchnum)
if ls != NULL
  return ls
rs = search(Rchild(btree), searchnum)
~~if rs != NULL~~
  return rs
~~return NULL~~
```

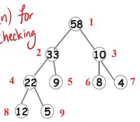
```
ADT BinTree
 BinTree create(bt);
 Boolean isEmpty(bt)
 BinTree MakeBT(bt1,
item, bt2);
 BinTree Lchild(bt);
 BInTree Rchild(bt);
 element Data(bt);
```

Max Heap Representation

- A complete binary tree can be represented with arrays
  - For any i-th node, $1 \leq i \leq n$,
    - parent(i): $\lfloor i/2 \rfloor$
    - leftChild(i): $2i$ if $2i \leq n$. Otherwise, no left child.
    - rightChild(i): $2i+1$ if $2i+1 \leq n$. Otherwise, no right child.
- A max *heap* can be represented as an array arr of elements such that
  - $arr[i] \geq arr[2i]$ if $2i \leq n$
  - $arr[i] \geq arr[2i+1]$ if $2i+1 \leq n$

O(n) for checking

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
58	23	10	22	9	8	4	12	5



Heap: Array Design

- Heap can be implemented with an array arr and a field capacity, a field size to record the number of elements
  - Create(capacity): allocate an array of size capacity+1.
    - heap.arr stores the allocated array.
    - heap.capacity=maximum number of elements that can be stored
    - Set heap.size=0.

Algorithm: *Create(capacity)*
```
heap <- allocate memory for heap
heap.arr <- allocate memory for arr with
size capacity+1
heap.capacity = capacity
heap.size = 0
return heap
```

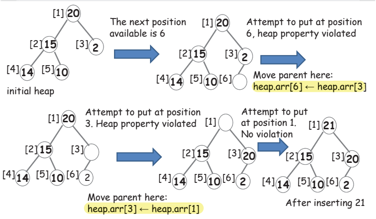
- isFull(heap)
  - Notice that we leave the heap.arr[0] as a place holder.

Algorithm: *IsFull(heap)*
```
return heap.size == heap.capacity
```

- isEmpty(heap)

Algorithm: *IsEmpty(heap)*
```
return heap.size == 0
```

Heap Insertion (to insert 21)



The next position available is 6

Attempt to put at position 6, heap property violated

Move parent here: heap.arr[6] ← heap.arr[3]

Attempt to put at position 3. Heap property violated

Attempt to put at position 1. No violation

Move parent here: heap.arr[3] ← heap.arr[1]

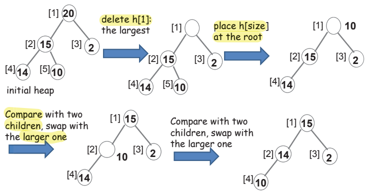
After inserting 21

Heap Insertion: Implementation

- Put it at the (size+1)-th position in the complete binary tree.
- But this may violate the heap property
  - Denote the current position as i. Its parent position is at $\lfloor i/2 \rfloor$
  - If the current position is at the root, just insert item to root
  - Otherwise, compare the parent and the item to be inserted.
    - If the parent has smaller key, move parent to current position, change the current position to i/2, and repeat above process.
    - Otherwise, insert the item to the current position

Algorithm: *InsertHeap(heap, item)*
```
if isFull(heap)
   error “Heap full”
heap.size = heap.size+1
i = heap.size
While i!= 1 and item.key > heap.arr[i/2].key
    heap.arr[i] = heap.arr[i/2]
    i = i/2
heap.arr[i] = item
```

Heap Deletion



delete h[1]: the largest

place h[size] at the root

Compare with two children, swap with the larger one

Compare with two children, swap with the larger one

Heap Deletion

Algorithm: *DeleteHeap(heap)*
```
item ← heap.arr[1] //copy the root data, which is to be deleted
lastitem ← heap.arr[heap.size] //copy the value of the last node
heap.size ← heap.size-1 //reduce the size of the heap by 1
i ← 1, child ← 2
While child <= heap.size
    if child<heap.size and heap.arr[child].key<heap.arr[child+1].key
       child ← child + 1 //choose the larger child
    if lastitem.key >= heap.arr[child].key
        break // no violation if we insert at position ‘parent’
    heap.arr[i]←heap.arr[child]//swap larger child with parent
    i ← child //attempt to insert at the child position
    child ← 2*child
heap[i] ← lastitem
return item
```

Search in BST: Implementation

- Return the node whose key is equal to the search key. If no such node exists, return NULL.
  - Solution: Divide-and-Conquer

左小右大

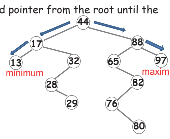
Unique key

Algorithm: *search(root, key)*
```
if isEmpty(root)  //base case: empty tree
   return NULL
data ← Data(root)
if key == data.key //equal to the search key, return the node
   return root
elseif key<data.key //larger than the search key,check left subtree
   rs=search(Lchild(root), key)
    return rs
else // smaller than the search key, only check the right subtree
   rs = search(Rchild(root), key)
   return rs
```

Minimum/Maximum in BST

- Minimum: Following the left child pointer from the root until the node has no left child.

Algorithm: *min(root)*
```
node = root
While !isEmpty(node)
and !isEmpty(Lchild(node))
   node = Lchild(node)
return node
```

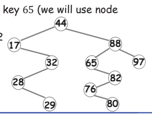


- Maximum: Following the right child pointer from the root until the node has no right child
  - Symmetric to minimum
  - Pseudocode: left as exercise

Successor in BST

[82] → left parent of 76

- Successor of a node x is the node with the smallest key greater than $x.key$
  - E.g.: the successor of the node with key 65 (we will use node 65 for short) is node 76
  - The successor of node 80 is node 82
- Two cases:
  - Node x has or has no right child



Algorithm: *successor(x)*
```
if !isEmpty(Rchild(x)) // case 1
  return minimum(Rchild(x))
xparent = parent(x)
While !isEmpty(xparent) and x ==
Rchild(xparent)  // case 2
   x = xparent
   xparent = parent(xparent)
return xparent
```

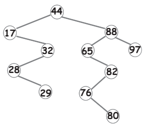
Case 1: return the minimum from the right subtree.

Case 2: if a node x is the right child of y, we call y to be the right parent of x, e.g., node 76 is the right parent of node 80. We repeatedly set the current node as the right parent of the current node until it has no right parent. Finally return the parent of the current node.

Predecessor in BST

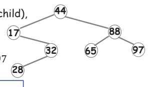
1. Check Child (L / R)
2. Check Parent (R / L)

- Predecessor of a node x is the node with the largest key smaller than $x.key$
  - The predecessor of node 32 is 29
  - The predecessor of node 76 is 65
- Also two cases
  - Node x has or has no left-child
- Implementation
  - Symmetric to successor
  - Left as self-exercise



Inorder traversal of BST

- What is the output if we feed a binary search tree T to Algorithm ***Inorder***? L D R
  - Let O_{17} be the output of left subtree of root node with key 44.
  - Let O_{88} be the output of right subtree of root node with key 44.
  - Then the output is: O_{17} 44 O_{88}
  - O_{17}: 17 O_{32} (since node 17 has no left child),
  - O_{32}: 28 32 $\Rightarrow O_{17}$ = 17 28 32
  - O_{88}: 65 88 97
  - The output is then: 17 28 32 44 65 88 97



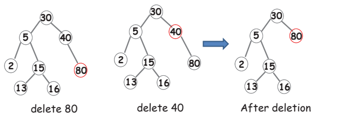
The Inorder traversal visits and outputs the nodes of a BST in increasing order of the keys!

Insertion of BST

- Base case: the root is empty, insert the data at the root.
- Otherwise, follow the search path until we find a node x such that
  - (i) $x.key = data.key$, simply return since BST does not allow duplicate keys
  - (ii) node x has no right child and $x.key < data.key$
    - Insert as the right child of x
  - (iii) the node has no left child and $x.key > data.key$
    - Insert as the left child of x
- Pseudocode: left as self exercise
- Time complexity?
  - $O(h)$, where h is the height of the BST

Deletion of BST: Cases 1 and 2

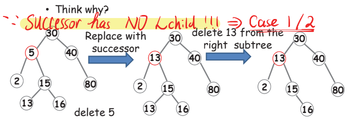
- Case 1: delete leaf node
  - If the leaf node is a right (resp. left) child, just set the right (resp. left) child of its parent to NULL
- Case 2: delete a node with a single child
  - Put the single child at the place of the deleted node



Deletion of BST: Case 3

- Case 3: delete a node with two children
  - We should replace a node here. But which node to put here?
  - The successor of the node to be deleted
    - The successor of node 5 is node 13
  - Replace the current node with its successor. Then delete the successor from the right sub-tree.
    - The deletion of successor falls into case 1 or case 2.
    - Think why?

∵ Successor has NO Lchild !!! ⇒ Case 1/2

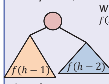


Height of Balanced Binary Search Tree

Theorem 1: Given a balanced binary search tree T of n nodes, the height, or equivalently the depth, of T is $O(\log n)$.

Proof: Let $f(h)$ be the minimum number of nodes of a balanced binary search tree of height h. Then, it is easy to verify that $f(1) = 1, f(2) = 2$.

For any $h \geq 3$, we have that $f(h) = f(h-1) + f(h-2) + 1$



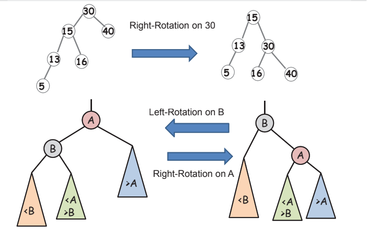
When h is even number:
$f(h) > f(h-1) + f(h-2)$
$> 2f(h-2)$
$> 4f(h-4)$
…
$> 2^{\frac{h}{2}-1} \cdot f(2) = 2^{\frac{h}{2}}$

When h is odd number:
$f(h) > f(h-1)$
$> 2^{\frac{h-1}{2}}$

Therefore, given a balanced BST of n nodes of height h, we have:
$n > 2^{\frac{h-1}{2}} \Rightarrow h < 2\log_2 n + 1 \Rightarrow h = O(\log n)$

Rotation Examples

Dynamic Rebalancing


30
15
40
13
16
5
Right-Rotation on 30
15
13
30
5
16
40
A
B
<B
<A >B
>A
Left-Rotation on B
Right-Rotation on A
B
A
<B
<A >B
>A