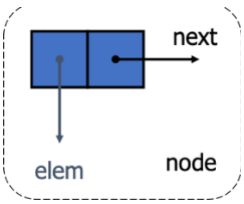
Arrays:

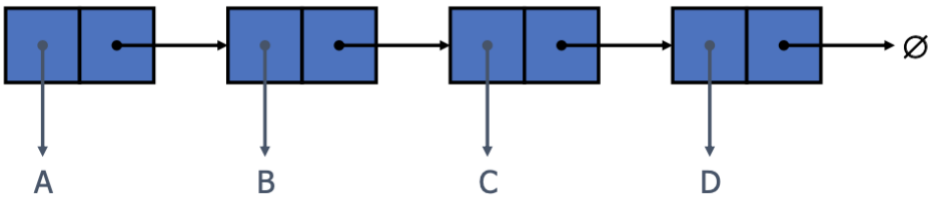
- data structure is a systematic way of organizing and accessing data
- An array is a sequence of locations in memory which stores a data point at every location
- Array size n-1
- Pointers:

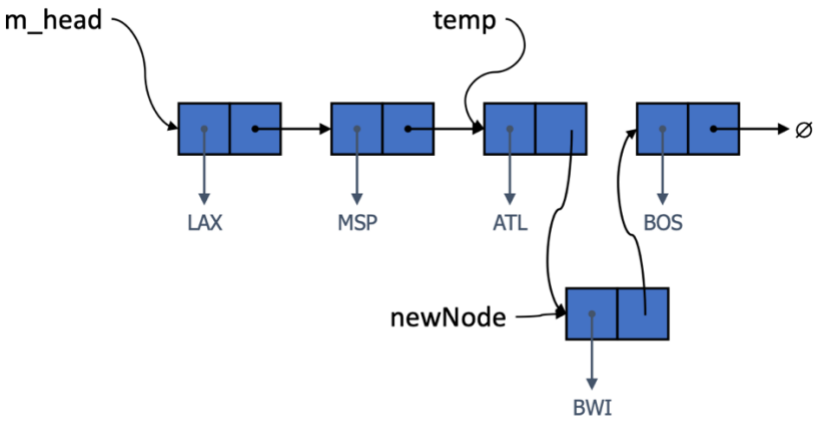
```
Object*   arrayName = new Object[9];
```

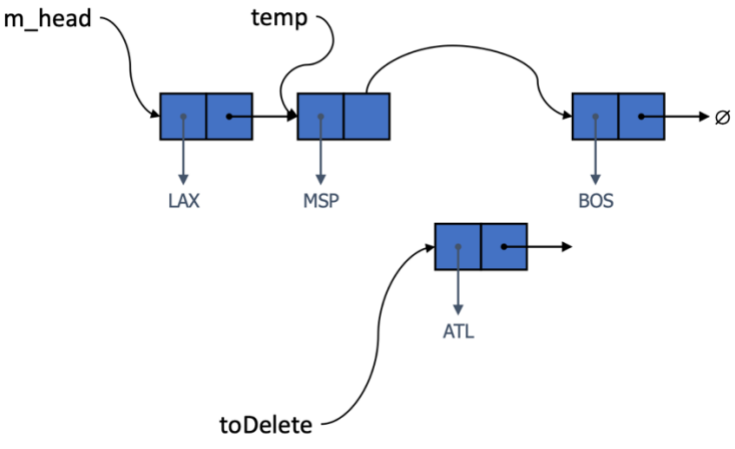
-
- Pre defined size
- array is a pointer to its first location
- insertion
- Assuming the list is full, i.e. 12 elements:
- To insert element at index 10, we only shift one element to right.
- To insert element at index 0,we shift 11 elements to right.
- Deletion
- Assuming the list is full, i.e. 12 elements:
- To insert element at index 10, we only shift one element to left.
- To insert element at index 0,we shift 11 elements to left.

Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
    - element
    - link to the next node
- 

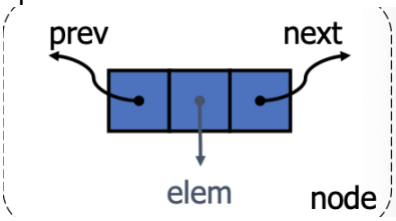
- 

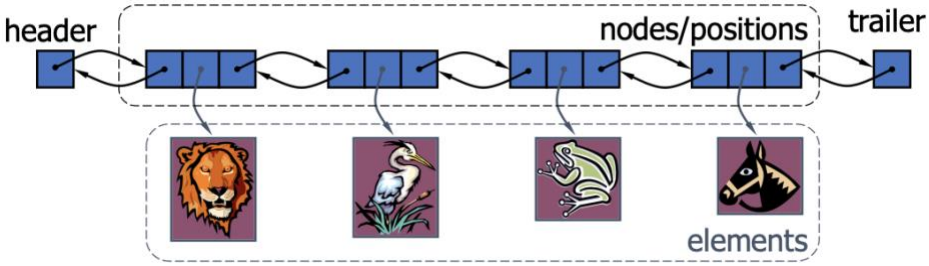
- Good to use when
    - we don't know the size of array before hand
    - keep a sequence of data points, but every sequence differs in the number of data points.
- Insert and remove
- no way to access individual elements directly
- insert:
- start from head and traverse using temp node
- when found create new node and re attach
- 

- Inserting at head = O(1)
- Inserting at tail = O(n)
- Deletion:
- start from head and traverse using temp node, stop before element to delete
- 

- Deletion at head = O(1)
- Deletion at tail = O(n)

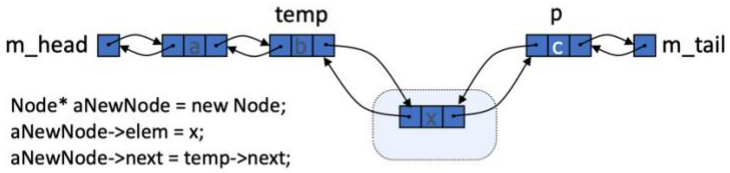
Doubly Linked List

- A doubly linked list provides an implementation of the Node List ADT
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes
- 

- 

- Insertion: look ahead, find the node, reconstruct with temp arrays

```
Node* temp = m_head;
while (temp!=m_tail && temp->next->elem != c) {temp = temp->next;}
```




```
Node* aNewNode = new Node;
aNewNode->elem = x;
aNewNode->next = temp->next;
aNewNode->prev = temp->next->prev
temp->next->pre = aNewNode;
temp->next = aNewNode;
```

- Inserting at head = O(1) if we start from the head
- Inserting at tail = O(1) if we start from the tail
- Worst case for insertion is O(n) depending where we start
- Memory usage will be double

Stack ADT

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations
- Stack ADT stores arbirtrary objects
- Insert/delete is LAST IN FIRST OUT
  - Push(), pop()
- Auxiliary Stack operation
  - Top, size, empty
- Applications:
  - Last page visited
  - Undo in text edit
  - Chain of method calls in c++ runtime system
  - ADS for algorithms
  - Components of other data structures
- Stack can be implemented in
  - Array
  - Linked List
- Array Stack:
- Insertion:

```
template <typename E>     // push element onto the stack
void ArrayStack<E>::push(const E& e) {
    if (size() == capacity) throw StackFull("Push to full stack");
    S[++t] = e;
}
```

- Primitive operation of insertion, O(1), count if statement = O(2)
- Or O(constant)
- Deletion:

```
template <typename E>     // pop the stack
void ArrayStack<E>::pop() {
    if (empty()) throw StackEmpty("Pop from empty stack");
    --t;
}
```

- Same as insertion
- O(constant)
- Space is O(n)
- the memory used by an algorithm is a computer resource used by the algorithm
- Linked List Stack:
- Insertion:

```
void LinkedStack::push(const Elem& e) { // push element onto stack
    ++n;
    S.addFront(e);
  }
```

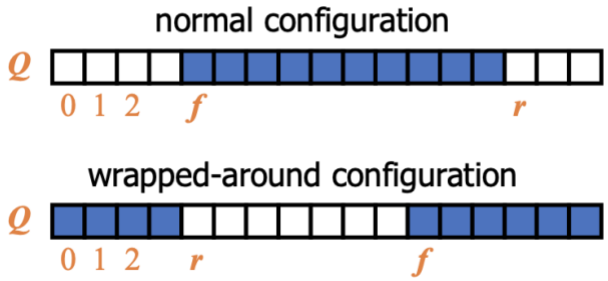
-
- O(1)
- Deletion:

```
void LinkedStack::pop() { // pop the stack
    if (empty()) throw StackEmpty("Pop from empty stack");
    --n;
    S.removeFront();
  }
```

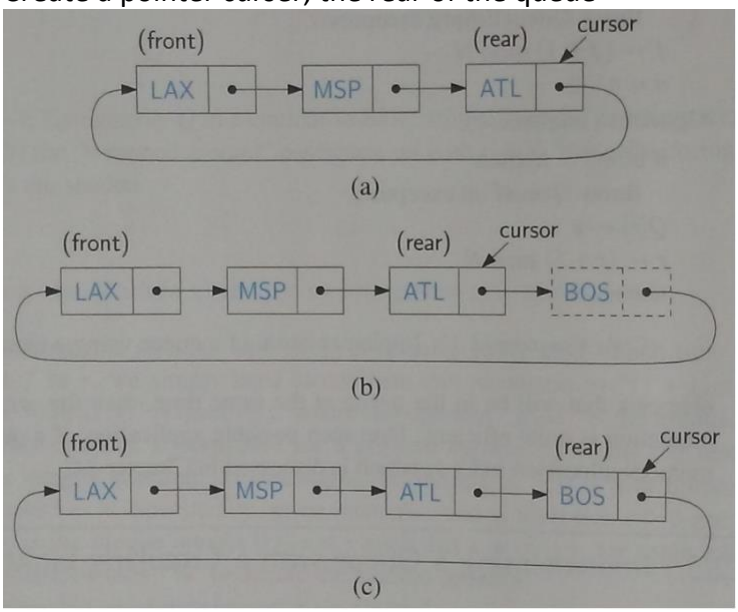
-
- O(1)

Queue ADT

- FIRST IN FIRST OUT
- Insert at rear, removal is front of queue
  - Enqueue, dequeue, front, size, empty
  - Throw exception at deletion or front of empty queue
- Application:
  - Waiting lists
  - Printers – access to shared resources
  - Operating system multitasking
- Implementation:
  - Array
  - Linked list
- Array Queue:
- deletion: normal would be O(n)
- Using circular array = O(1)
- Front index, index past the rear element, number of items in queue

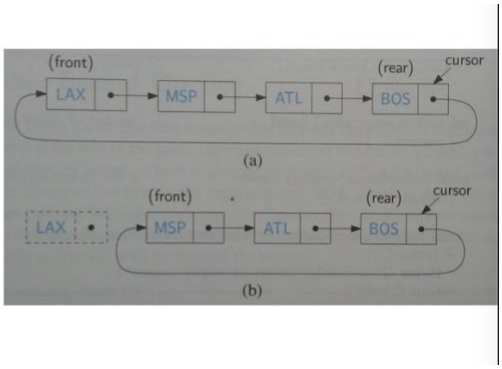


-
- Index+1%array_size = location of rear element
- Linked List Queue:
- Insertion:
- Normal way would be O(n)

- Circular = O(1)
- Create a pointer curser, the rear of the queue
- 

- Deletion:

  Dequeue,
  1 – Copy the next pointer of the cursor (front item) into a temporary node.
  2 – Copy the next pointer of the front item into the next pointer of the cursor.
  2 – Delete the temporary node (front item)

-

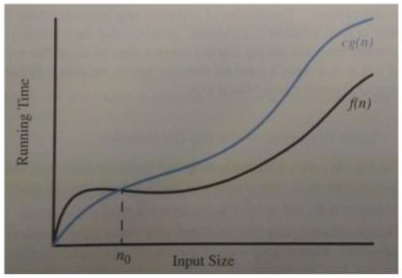
Asymptotic Analysis

- Running time vs size of input = asymptotic analysis
- Primitive count is f(n) < Big O(g(n))
- (6n – 1) <= cn for c = 6 and n sub 0= 1
- Holds true for any c >=6, n sub 0>= 1

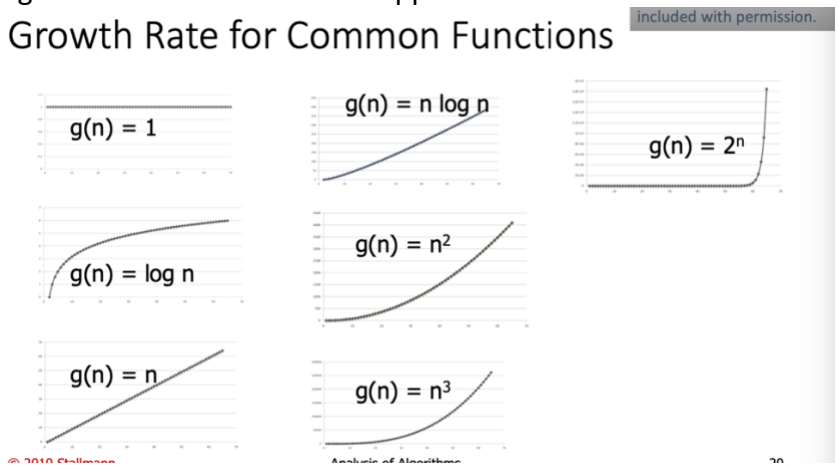
Big-O Notation

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants $c > 0$ and $n_0 \geq 1$ such that

$$f(n) \leq cg(n) \text{ for } n \geq n_0$$



- ---
- Big O = worst case scenario = upper bound



- Find the upper bound for the function (2n + 100 log n).
  Note: log n = $\log_2 n$

  For $n \geq 2$, we can say $(2n + 100 \log n) \leq (2+100) n$, since n is the highest order, therefore,

  $(2n + 100 \log n)$ is O(n), where c = 102, and $n_0 = 2$

$2^{n+2} = 2^n * 2^2 = 4 * 2^n$

(2^{n+2}) is $O(2^n)$, where $c = 4$, and $n_0 = 1$

- Find the upper bound for the function ($\log n^3$).
  Note: $\log n^3 = \log_2 n^3$
  Using the logarithm rule $\log_b(a^c) = c*\log_b(a)$, we can say

  $\log n^3 = 3 \log n$

  $(\log n^3)$ is $O(\log n)$, where $c = 3$, and $n_0 = 2$, please note, for $n = 1$, $\log n$ would be zero.
- $$1 + 2 + 3 + \ldots + (n-2) + (n-1) + n = n(n+1)/2$$
- Summation rule

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

- Nested loops is where O(n^2) occurs

$1 + 2 + \ldots + (n-1) = (n-1)(n-1+1)/2 = n(n-1)/2$

$f(n) = 4n + 2\,(n(n-1)/2) + 1$
$= 4n + n^2 - n + 1$
$= n^2 + 3n + 1$

$n^2 + 3n + 1 \leq (1+3+1)n^2$
$n^2 + 3n + 1$ is $O(n^2)$, where $c = 4$, and $n_0 = 1$

- Big Omega – Best case scenario/ lower bound

  - **big-Omega**
    - f(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c\, g(n)$ for $n \geq n_0$
- Big Theta – Big omega and big O are equal best = worst
  - **big-Theta**
    - f(n) is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c'\, g(n) \leq f(n) \leq c''\, g(n)$ for $n \geq n_0$
- Big omega – find the function that is smaller in polynomial

Find the lower bound for the function (3log n + 2n).

$3\log n + 2n \geq 3\log n$

$(3\log n + 2n)$ is $\Omega(\log n)$, where $c = 3$, and $n_0 = 2$

Note: for $n_0 = 1$, $3\log n = 0$ ($\log 1 = 0$)

- Find the average bound for the function (3nlog n + 4n + 5log n).

  $3n\log n \leq 3n\log n + 4n + 5\log n \leq (3+4+5)n\log n$

  $(3n\log n + 4n + 5\log n)$ is $\Theta(n\log n)$, where $c' = 3$, $c'' = 12$ and $n_0 = 2$

  Note: for $n_0 = 1$, $3n\log n = 0$, and $n\log n = 0$ ($\log 1 = 0$)

constant	logarithm	linear	n-log-n	quadratic	cubic	exponential
1	log n	n	n log n	n^2	n^3	a^n

Amortized Analysis

- That is the average number of operations per data element,
- = total cost of operations/n
- Use this in rebalancing

- The average cost of insertion is (n + 1)/2 if we resize the array by one every time.

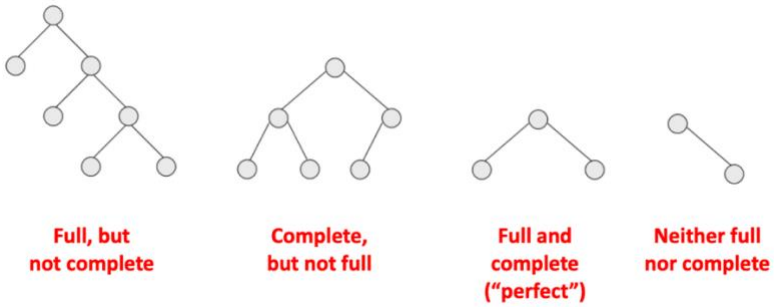
  The average cost of insertion is 2 if we double the array size every time.

  (n + 1)/2 is in terms of data size n, and 2 is a constant running time.

  Therefore, the algorithm which doubles the array size is much more efficient.

Tree ADT

- Tree – collection of node elements
- Depth : start at head = 0 count each child +1 till leafs
- Height: edges between root and farthest leaf
- Binary Trees:
- Application:
    - Arithmetic expressions
    - Decision process
    - Searching
- Full: every node has 0 or 2 children (not one child)
- Complete: Every leaf on same level, leaves filled in left to right (node doesn't need a right child leaf)
- Full: every lead is on same level, adding another node = new level



-
- Binary tree implementation:
    - Linked list
    - Array based
- BST = insert new node, value less than parent on right, value greater than parent on left
- Relationship in perfect tree, num of node = 2^level
- Tree traversal – recursion
    - *Preorder*: Visit root, traverse left, traverse right
    - *Inorder*: Traverse left, visit root, traverse right (Free fall)
    - *Postorder*: Traverse left, traverse right, visit root (break the leaves left, right)

3. Visit the root.
4. Preorder traverse the left subtree.
5. Preorder traverse the right subtree.

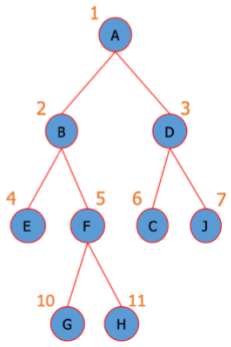
3. Inorder traverse the left subtree.
4. Visit the root.
5. Inorder traverse the right subtree.

3. Postorder traverse the left subtree.
4. Postorder traverse the right subtree.
5. Visit the root.

-
- Array based implementation

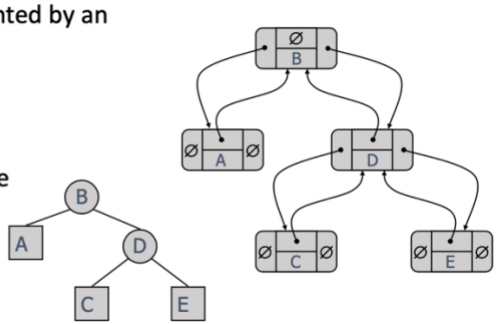
Array-Based Representation of Binary Trees

- Nodes are stored in an array A



- Node v is stored at A[index(v)]
  - index(root) = 1
  - if node is the left child of parent(node), index(node) = 2 * index(parent(node))
  - if node is the right child of parent(node), index(node) = 2 * index(parent(node)) + 1

-
- Doubly linked list BST
  - A node is represented by an object storing
    - Element
    - Parent node
    - Left child node
    - Right child node



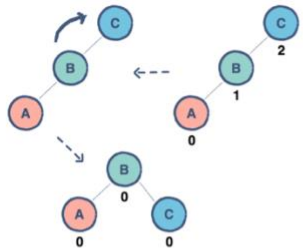
-
- BST deletion
  - Case NO CHILDREN: traverse to node and delete
  - Case ONE CHILD: Copy child data into node toDelete vise versa and then delete
  - Case TWO CHILD: Copy right childs min value aka the left grandson, then delete
- Worst case: O(n) when tree is a “linked list”
- Balanced tree: O(log n)

AVL Tree

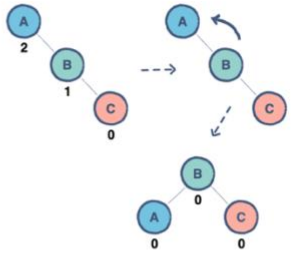
- Keeping our BST tree always balanced
- Balanced: Hight of left – right child is at most 1
- We do rotations, if there is a left child to a a rotation, remember magenta is right child of that new node

Case	Balancing Operations
Node's right child has greater height and right child of right child has a greater height	Left rotation
Node's right child has greater height and left child of right child has a greater height	Right rotation about node's right child, left rotation about node
Node's left child has greater height and left child of left child has a greater height	Right rotation
Node's left child has greater height and right child of left child has a greater height	Left rotation about node's left child, right rotation about node

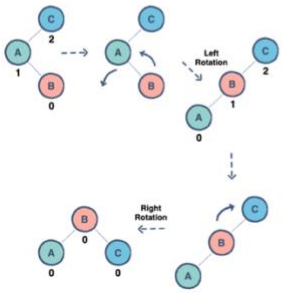
- Right Rotation
  Right around C



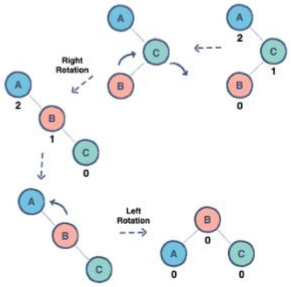
Left Rotation:
Left around A



Left-Right rotation
Left around B
Right around C



Right-Left rotation
Right around C
Left around A



- Height of AVL Tree is:

The height of AVL tree is O(log n), where n is the input size.

- n is the number of internal nodes
- we do not store data in the leaves (leaf nodes are null pointers)*

AVL Tree Performance

- a single restructure (left or right rotation) takes O(1) time
  - using a linked-structure binary tree
- ***find*** takes O(log n) time
  - height of tree is O(log n), no restructures needed
- ***insert*** takes O(log n) time
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)
- ***remove (erase, or delete)*** takes O(log n) time
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)

- Therefore, ***h is O(log n)*** for $n_0 = 3$ and $c = 2$

- We Rebalance after inserting or removing from tree,

Hight of an AVL tree

$n(h) > 2 * n(h-2)$

$n(h) > 4 * n(h-4)$

$n(h) > 8 * n(h-6)$

- Where n(h) = internal nodes, h = height, I =h/2 -1

Priority Queue ADT and Heap

- Most important, first out
- Ex, emergency rooms, airline boarding, auctions
- Key associated to determine priority
- Comparable value generated based on requirements
- NOTE: allowed to have multiple entries with same key value
- Insert, removeMin/Max, min/max, size, empty
- Using a sort list, remove min= O(1) but cost to sort the list

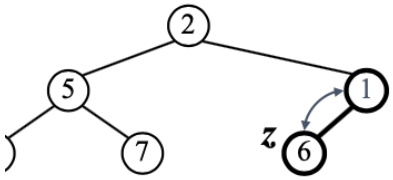
Operation	Unsorted List	Sorted List
size, empty	O(1)	O(1)
insert	O(1)	O(n)
min, removeMin	O(n)	O(1)

- C++ STL provides a priority queue which by default it returns the max value in the queue. Can specify min, std::greater<int> < q
- THE HEAP OF PRIORITY Q
- Heap is BINARY TREE NOT BST
- Heap conditions:
  - Nodes sorted based on relation to parents
  - Ordered based on key
  - Tree Must be Complete: every leaf is on the same level, all leaved filled left to right, no need for right child at end.
- Hight of heap = O(log n): At depth (h-1) there are 2h-1 nodes
- Min Binary Heap:
  - Complete tree
  - No child is smaller than parent
  - Both children >= parent
- Partial ordering: values grow as we increase distance from root
  - path from the root to a leaf visits nodes in a non-decreasing order
- Convert Heap to array: root is index 1
  -

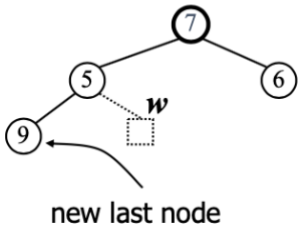
To Find	Formula
Parent index	`floor((index)/2)`
Left Child index	`2(index)`
Right Child index	`2(index) + 1`

- Keep track of last node z
- Insertion:
  - Find the insertion node ***z*** (the new last node)
  - Store ***k*** at ***z***
  - Restore the heap-order Bottom up

- Restoring the heap property:
  - Swap k along an upward path from insertion
  - Terminate when k reaches node or parent < k
  - Restoring heap runs O(log n)



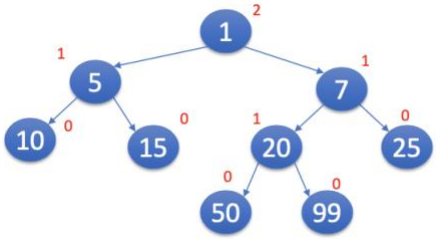
-
- Removal from heap
  - Replace the root key with the key of the last node w
  - Remove w
  - Restore the heap-order This time from top down



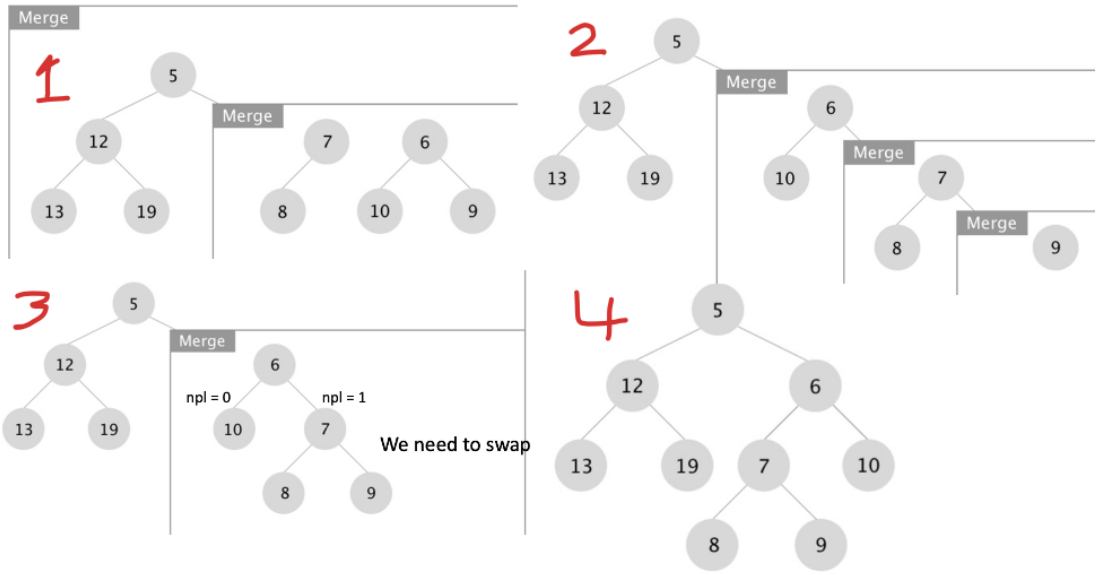
-

Leftist Heap

- If we try to merge two heaps together:
  - Insertion is O(log n) + n elements into heap = O(n log n) worst case
- We can do better
- Leftist Heap:
  - Insertion and deletion use merge operation
  - Binary Tree with heap property
  - Null Path Length – shortest path to null
  - NPL value of right child = or less than left child
    - Right side of leftist has shortest path
    - Left heavy tree
    - Merge work done on right side
- Null Path length (npl): shortest path to node without exactly 2 children
  - npl(null) = -1
  - npl(leaf) = 0
  - npl(single-child node) = 0
  - npl(x) = min distance to a descendent with 0 or 1 child

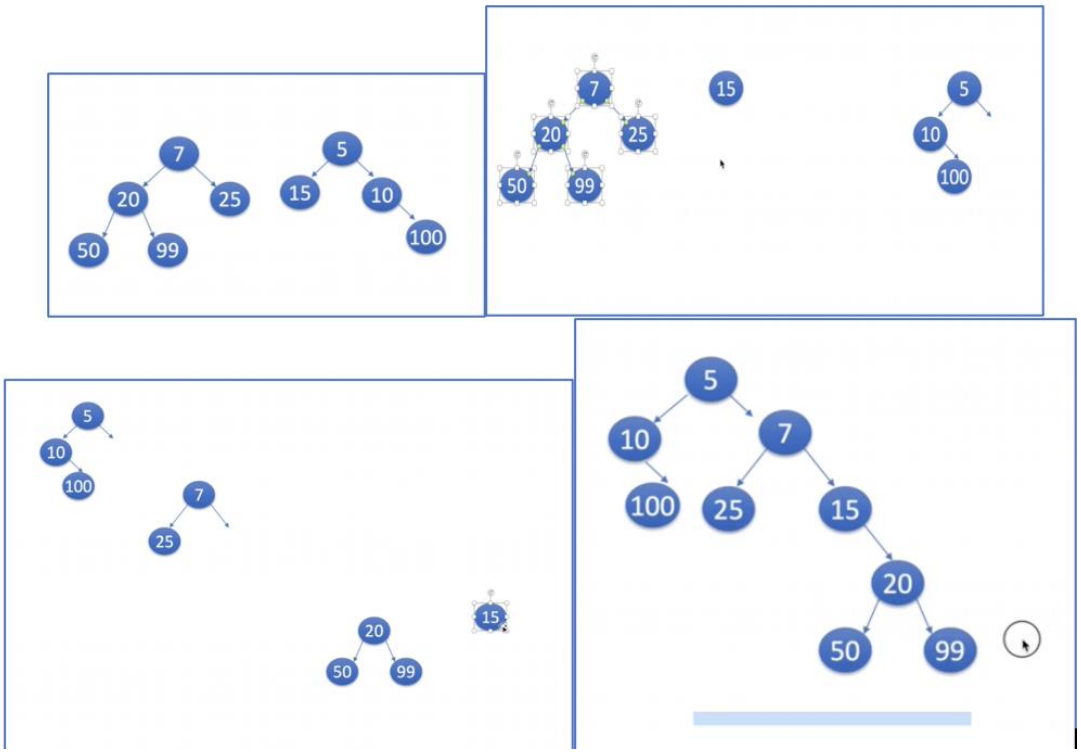


- Npl at every node is the min npl of two children + 1
- How to merge:
  - Compare two roots: take apart right subtree of smallest root
  - Compare that recursively until Reached null
  - Zip back up and check npls NPL value of right child = or less than left child
  - Swap subtrees if so
- Insertion: single node treated as its own tree merge with existing heap
- Deletion: remove the top
  - Left over is two sub heaps – merge them
- Insertion, deletion, merging amortized O(log n)



Skew Heap:

- Scientists noticed, right side of leftist often has greater npl then left = always swapping
- Why not swap every time we merge - Insertion, deletion, merging amortized O(log n)
- Basically after comparing the roots, swap the children of smaller parent, then compare
- In deletion:

- 

Splay Tree:

- If we search for a node in BST then we may need it the next time
- Bring that node up to root
- Node is splayed after search or update
- Bring last accessed node to root = splay
- Splaying cost O(h) h = height of tree
    - Worst case “linked list tree” = O(n)
    - Insertion, deletion, finding amortized O(log n)
- Splay uses rotation

a right rotation about y

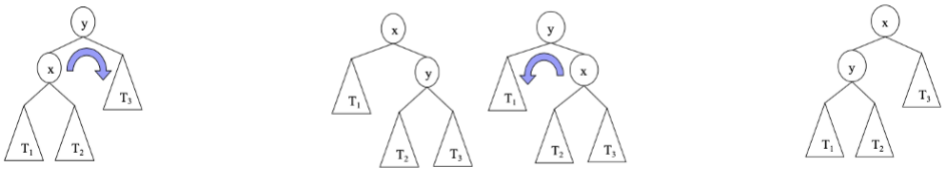
makes the left child x of a node y into y's parent; y becomes the right child of x

structure of tree above y is not modified

a left rotation about y

makes the right child x of a node y into y's parent; y becomes the left child of x

structure of tree above y is not modified



Before rotation | After rotation | Before rotation | After rotation

- Right rotation: Zig, Left rotation: Zag

Case – To find the key	Algorithm (Terminology)
we go right then left	Zig-Zag - Right rotation on parent, left rotation on grandparent
we go left then right	Zag-Zig - Left rotation on parent, right rotation on grandparent
we go left then left	Zig-Zig - Right rotation on grandparent, right rotation on parent
we go right then right	Zag-Zag - Left rotation on grandparent, left rotation on parent
From root we go left	Zig - Right rotation on root
From root we go right	Zag - Left rotation on root

- Different cases for splaying after each operation

method	splay node
find(k)	if key found, use that node if key not found, use parent of ending external node
insert(k)	use the new node containing the entry inserted
remove(k)	use the parent of the internal node that was actually removed from the tree (the parent of the node that the removed item was swapped with)

Map ADT And hash:

- Access by keys- values pairs
- Want to work in O(1)

Running Time

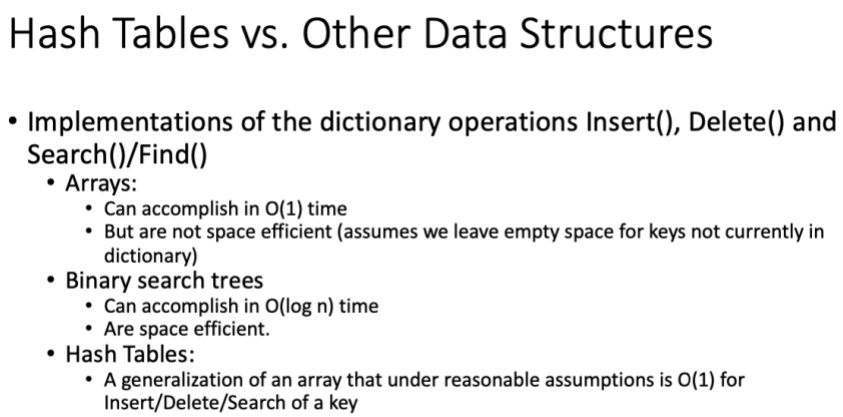
If we store data in an array, it will cost O(n), and look for a record by traversing the items sequentially, the search can cost O(n) in worst case.

If we use index numbers to access data points, it costs O(1). However, how a simple integer key can help us to find data?

If we store the data in a binary search tree, it will cost O(n), and a search will cost O(log n) in the worst case.

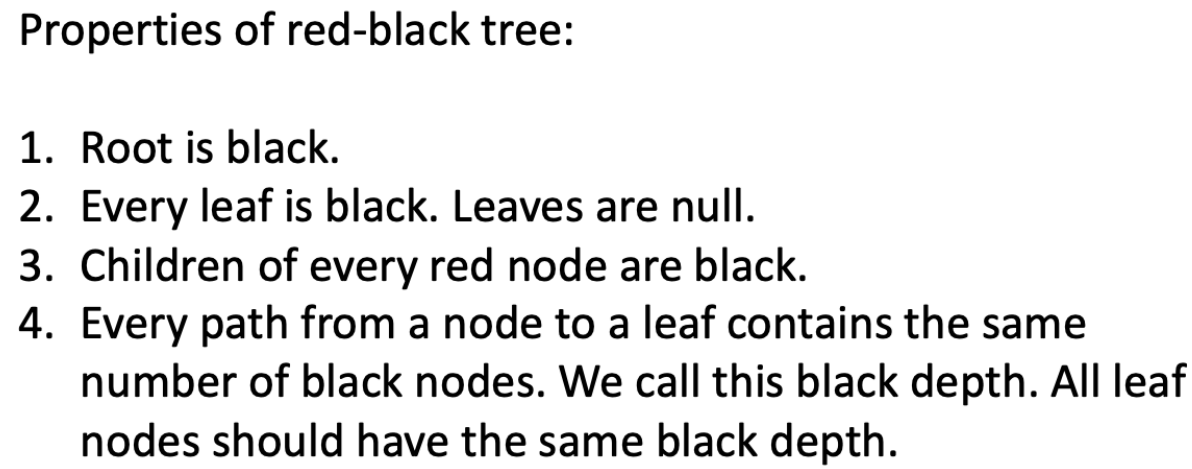
- Hash buckets arrays
- Hash functions transform key into index

- Returns the index of location
- Insert,delete,search in O(1)



-
- Collision occurs when two keys have same key
- Can implement linked list, can waste space, not as fast if they all end up there
-

Red and black tree



-
- Height O(log n)
- A lot of recollering and restructuring

A regular BST may become a linked list, which degrades the performance significantly.

A red-black tree is a balanced tree which does not need a lot of restructuring.

An AVL tree requires a lot of restructuring, which can affect the performance.

-

Dictionary ADT, Skip lists

- Dictionary stores the key values just like a map

  Linked lists:

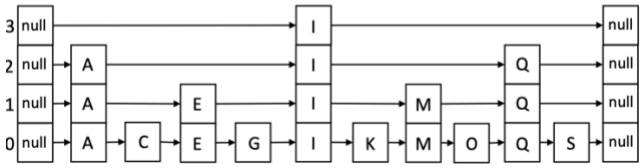
  Insert in O(1)
  No need to know the space in advance

  Search in O(n)
  Cannot jump to the middle

-

  Skip Lists

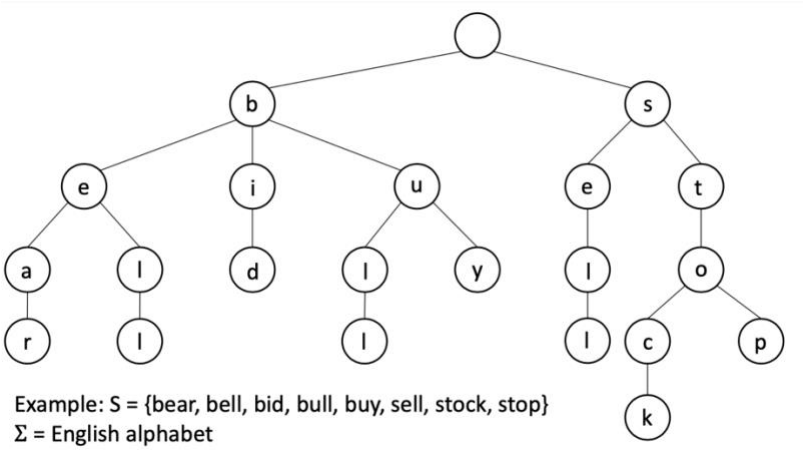
  Data is organized in multiple linked lists,
  Data is sorted.



-
- Search, insert, deletion time O(log n)
- Keeping it organized costs time
- Random values assigned is better
- Its similar to quad node
- Space used O(n)

Tries

- Storing strings allows for fast search
- Main goal information retrieval
- Searching data base



Example: S = {bear, bell, bid, bull, buy, sell, stock, stop}
Σ = English alphabet

- Search time is constant O(md) or O(d)
- Creating a node is O(1) then O(m) m being the size of string for insertion
- Insertion costs O(n)

Insertion:

To insert a string into the tree, we start to search and follow the path, we continue the search until it is not possible to continue, then we start to create a new chain of nodes.

If the path for all characters exist except the last one, the search time would be O((m-1)d). The the worst case of insertion is O(d).

Constructing the tree would be O(dn), where n is the total length of all strings. We can say the construction costs O(n).

K-d Trees

- Searching with multiple keys like on a map grid
- Nearist neighbor search
- Insertion divides the grid to two parts
- One responsible for x and y axis
- BST tree in which nodes are in k dimentional space
- Like a BST the worst-case running time of search operation in k-d tree can be O(n), and the average case is O(log n).

Graph ADT

A graph represents the relationships between pairs of objects.

Examples:

Computers on a network
You and your friends in a social media
Cities on a map connected with roads

-
- Google maps uses this, shortes route
- Network routing for shortest path to computers

A ***directed*** graph is the one in which all edges work only in one direction.

If the edges work in both directions the graph is called ***undirected*** graph.

We can assign a value to every edge for some property. For example, on a road map, every road between two cities represents the distance between them. We call such a graph a ***weighted*** graph.

-

Edge List
Adjacency Matrix
Adjacency List

-

Adjacent Nodes:
Two vertices are adjacent if there is an edge connecting them.

In this example,
Nodes 0 and 1 are adjacent.
Nodes 0 and 6 are adjacent.
Nodes 0 and 8 are adjacent.
Nodes 0 and 9 are not adjacent.

-

Performance Summery

constant	logarithm	linear	n-log-n	quadratic	cubic	exponential
1	log n	n	n log n	n^2	n^3	a^n

LAST IN FIRST OUT	Stack ADT – Array	Stack ADT – Linked List
Insert	O(1)	O(1)
Delete	O(1)	O(1)
Size in memory	O(n)	O(n)+ pointers
Advantage	Simplicity	Unknown size of data
Disadvantage	Cant resize Overflow Waste array space (unless C++ STL stack class)	Requires pointers = more space

Queue ADT – FIRST IN FIRST OUT	Array	Linked List	Doubly Linked List	Circular array	Circular linked list
Insert	O(1)	O(n)	O(1)	O(1)	O(1)
Delete	O(n)	O(1)	O(1)	O(1)	O(1)

- AVL Tree Performance

AVL Tree Performance

- a single restructure (left or right rotation) takes O(1) time
  - using a linked-structure binary tree
- ***find*** takes O(log n) time
  - height of tree is O(log n), no restructures needed
- ***insert*** takes O(log n) time
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)
- ***remove (erase, or delete)*** takes O(log n) time
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)

Runtime Analysis for BST and AVL Tree

BST	Average Case	Worst Case
Insert	O(logn)	O(n)
Delete	O(logn)	O(n)
Search	O(logn)	O(n)

AVL	Average Case	Worst Case
Insert	O(logn)	O(logn)
Delete	O(logn)	O(logn)
Search	O(logn)	O(logn)

Operation	Unsorted List	Sorted List
size, empty	O(1)	O(1)
insert	O(1)	O(n)
min, removeMin	O(n)	O(1)

Red-Black Tree Operations Running Time

Operation	Running Time
size	O(1)
empty	O(1)
find	O(log n)
insert	O(log n)
remove	O(log n)

The height of Red-Black Tree is O(log n).

Leftist Heap Complexity

Operation	Worst-case
MakeHeap	O(1)
FindMin	O(1)
Union	O(log n)
Insert	O(log n)
DeleteMin	O(log n)
DecreaseKey	O(log n)
Delete	O(log n)

Skew Heap Complexity

Operation	Amortized
MakeHeap	O(1)
FindMin	O(1)
Union	O(log n)
Insert	O(log n)
DeleteMin	O(log n)
DecreaseKey	O(log n)
Delete	O(log n)

Performance Summery

	Singly Linked List	Doubly Linked List	Array
Insert at head	1	1	n
Insert at tail	n	1	1
Delete from head	1	1	n
Delete from tail	n	1	1
Insert at a location	n	n	n
Delete at a location	n	n	n
Find a node	n	n	1

Stack Operations Running Times

Operation	Array Implementation	Linked List Implementation
Push	O(1)	O(1)
Pop	O(1)	O(1)
Top	O(1)	O(1)
Size	O(1)	O(1)
Empty	O(1)	O(1)

	Singly Linked List	Doubly Linked List
Insert at head	1	1
Insert at tail	n	1
Delete from head	1	1
Delete from tail	n	1
Insert at a location	n	n
Delete at a location	n	n
Find a node	n	n

Operation	Unsorted List	Sorted List
size, empty	O(1)	O(1)
insert	O(1)	O(n)
min, removeMin	O(n)	O(1)

Running time for implementing a priority queue using a heap

Operation	Unsorted List
size, empty	O(1)
min	O(1)
insert	O(log n)
removeMin	O(log n)

The space used is O(n).

Hash Tables vs. Other Data Structures

- Implementations of the dictionary operations Insert(), Delete() and Search()/Find()
  - Arrays:
    - Can accomplish in O(1) time
    - But are not space efficient (assumes we leave empty space for keys not currently in dictionary)
  - Binary search trees
    - Can accomplish in O(log n) time
    - Are space efficient.
  - Hash Tables:
    - A generalization of an array that under reasonable assumptions is O(1) for Insert/Delete/Search of a key

Red-Black Tree Operations Running Time

Operation	Running Time
size	O(1)
empty	O(1)
find	O(log n)
insert	O(log n)
remove	O(log n)

The height of Red-Black Tree is O(log n).

Tries Operations Running Time

Operation	Running Time
Search	O(d), constant time
Insert	O(d), constant time
Remove	O(d), constant time
Constructing	O(n)

Note: d is the alphabet size.

Tries space is O(n).

Operations Running time

Operation	Running Time (Average)
Insert	O(log n)
Search	O(log n)
Remove	O(log n)

Space of k-d tree is O(n).

DFS Traversal Running time

The running time of DFS traversal is O(n+m) in which n is the number if vertices (nodes) and m is the number of edges.

We know that we only visit a node once, then we tag it as visited. Then this is n operations.

For every node we need to check all its edges to its adjacent nodes. This means we are checking on all edges once; this is m operations. This is done to see whether we already visited the adjacent node.