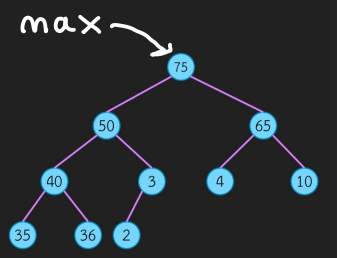
Data Structure Summary

Data Structure	Pro	Cons	Remark
pair<T1,T2>	Nothings... It just a pair of two data type		
vector<T>	• Fast access [ ] • Fast append	• Slow find • Slow insert, Slow Erase	
set<T>	• Fast find • Item is sorted	• Slower to just append data than vector, stack, queue • Iterate is also slow • Takes lots of memory	Require comparator
map<Key,T>			• Associative data type • Also require comparator of Key_Type
stack<T>	• Very fast push, pop	• Very limited functionality but has special uses	• No iterator • Order of data coming out depends on something (stack, queue depends on WHEN it is pushed, pq depends on value)
queue<T>			
priority_queue <T>	• Fast get max • Fast delete max • Data is sorted • Memory efficient	• Slower to just append data than vector, stack, queue • Very limited functionality	• PQ requires comparator

more data structure

- C++ has more data structure not really covered right now
  - list is a vector with faster insert / erase but does not have fast access
  - unordered_set, unordered_map are set and map that the data is not sorted but is much faster
  - deque (pronounced DECK) is a queue that can push, pop at both ends
  - multiset, multimap are set and map that allows duplicate entries

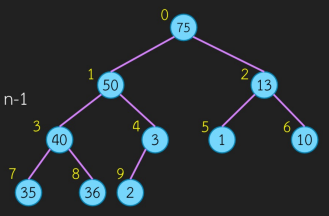
Binary Heap



- We use Complete Binary Tree to store data
  - A value is stored at the node
- When data is modified (via push or pop), we must maintain these rules
  1. Tree must always be Complete Binary Tree
  2. For any node, its value must be more than that of its children

cp::Priority_queue with binary heap

How to store a tree?

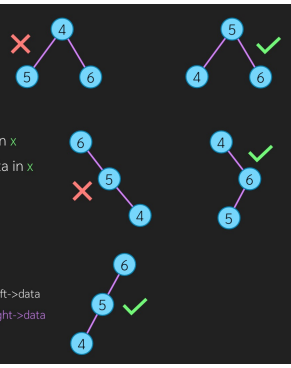


- Use dynamic array
  - Each node can be labelled from 0 to n-1
- Root is at 0
- Left child of node i is at (i*2)+1
- Right child of node i is at (i*2)+2
- Parent of node i is at (i-1)/2

0	1	2	3	4	5	6	7	8	9
75	50	13	40	3	1	10	35	36	2

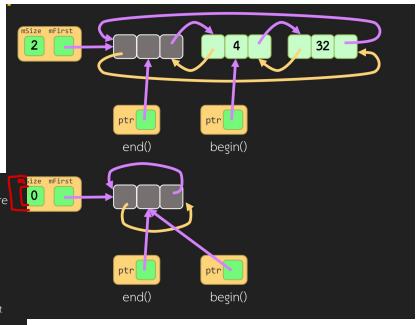
cp: map_bst

Binary Search Tree



- Structure rule: must be a Binary Tree
- Value rule: for any node x
  - data in left-subtree must be less than the data in x
  - data in right-subtree must be more than the data in x
- Recursive Definition
  - An empty tree is a Binary Search Tree (BST)
  - A node X is a BST when
    - Its subtrees (if any) must be BST and
    - If left-subtree exists, X->data must be more than x->left->data
    - If right-subtree exists, X->data must be less than x->right->data

cp::linked list

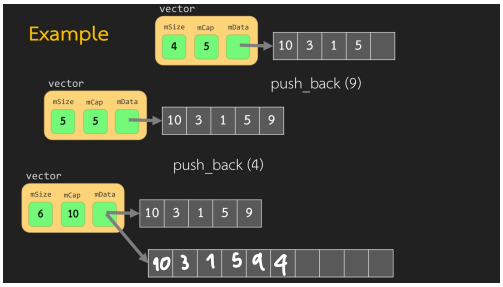


cp:: vector

Key Idea

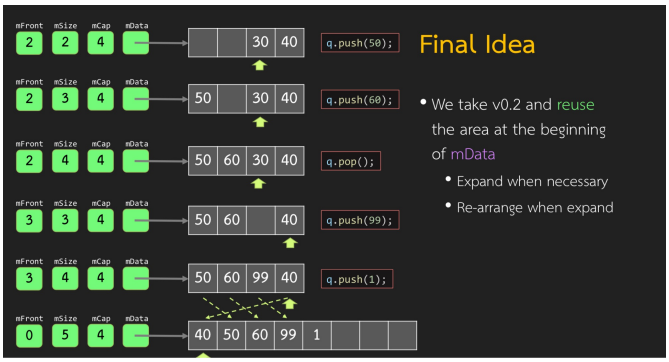
- Vector stored 3 things (3 member data)
  - mData: A dynamic array, large enough to store current data and might have reserved space
  - mSize: Number of data stored
  - mCap: Size of the dynamic array (maybe larger than mSize)
- If the dynamic array is full and more data is being added, we create a new dynamic array and relocate data to the new array
  - This is called expand
  - Each expansion takes very long time
- Dilemma
  - large reserve = less often relocation but use more memory
  - Small reserve = more frequent relocation but less memory

Example



cp::queue

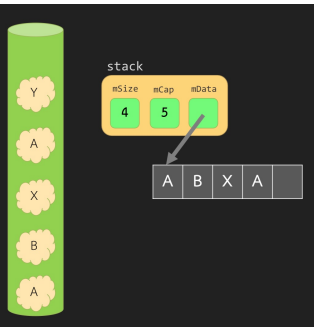
Final Idea



- We take v0.2 and reuse the area at the beginning of mData
  - Expand when necessary
  - Re-arrange when expand

cp: stack

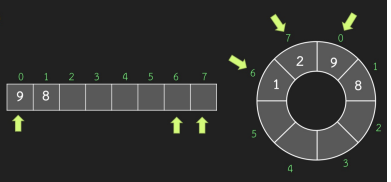
Key Idea



- The data is stored in the same way as a vector
  - The first element of mData is the bottom of stack while the last element is the top of stack
- We just take vector.h and remove unnecessary function

Circular queue

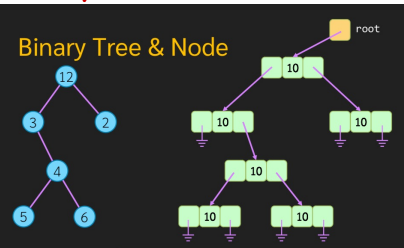
Circular Queue



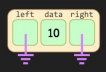
- We can think of mData to be circular
  - End of the last element of the mData is connected to the first element
- Consider i^{th} element
  - the next element is `(i+1) % mCap`
  - The previous element is `(i-1+mCap) % mCap`
  - Next k element is `(i+k) % mCap`

cp:binary tree

Binary Tree & Node

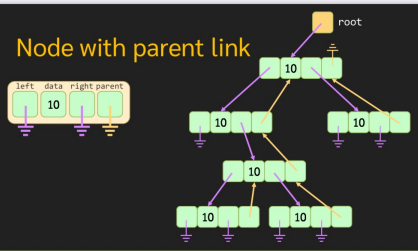


- A rooted tree where each node have at most two children
- Tree Node is very similar to a linked list node




```
class node {
  public:
    ValueT data;
    node  *left, *right;
    node() :
      data( ValueT() ), left( NULL ), right( NULL ) { }
    node(const ValueT& data, node* left, node* right) :
      data ( data ), left( left ), right( right ) { }
};
```

Node with parent link

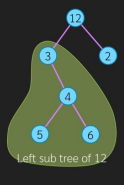


- Sometime, we need a link to parent
- Root is the only node that parent is NULL

```
class node {
  public:
    ValueT data;
    node  *left, *right, *parent;
    node() :
      data( ValueT() ), left( NULL ), right( NULL ), parent( NULL ) { }
    node(const ValueT& data, node* left, node* right, node* parent) :
      data ( data ), left( left ), right( right ), parent( parent ) { }
};
```

Sub tree

Subtree



- For any node
  - its left (right) child and all of the child's descendants is called left-subtree (right-subtree)

AVL_Tree

balanced binary serach tree

หมุนเมื่อไม่สมดุล

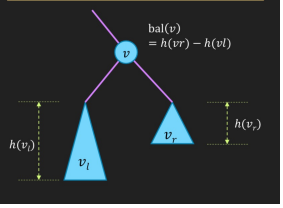
The Balance Constraint

- For each node v
  - We define v_l and v_r as the left and right subtree of v
  - We define $h(x)$ as the height of the tree x
    - Recall from that an empty tree has height as -1
  - We define a balance value of the node v as $\text{bal}(v)$
    - $\text{bal}(v) = h(v_r) - h(v_l)$
- the balance constraint is that $|\text{bal}(v)| \leq 1$
- Every node in the AVL tree must satisfy the balance constraint

```
class node {
  public:
    int data;
    node *left, *right;
};

int get_height(node *n) {
  if (n == NULL) return -1;
  return 1 + std::max(get_height(n->left),
                      get_height(n->right));
}

int balance_value(node *n) {
  if (n == NULL) return 0;
  return get_height(n->right) - get_height(n->left);
}
```



Rotation Summary

- Rebalance according to the balance value of the node

```
node * rebalance(node * r) {
  if (r == NULL) return r;
  int balance = r->balance_value();
  if (balance == -2) {
    if (r->left->balance_value() == 1)
      r->set_left(rotate_right_child(r->left));
    r = rotate_left_child(r);
  } else if (balance == 2) {
    if (r->right->balance_value() == -1)
      r->set_right(rotate_left_child(r->right));
    r = rotate_right_child(r);
  }
  r->set_height();
  return r;
}
```

Balance Value	Case	Example	Fix
-2 (Left Heavy)	Balance Value of Left Child is -1		Rotate Right at the node
	Balance Value of Left Child is +1		Rotate Left at the left child then Rotate Right at the node
+2 (Right Heavy)	Balance Value of Right Child is +1		Rotate Left at the node
	Balance Value of Right Child is -1		Rotate Right at the right child then Rotate Left at the node

Example Sequence of Operation