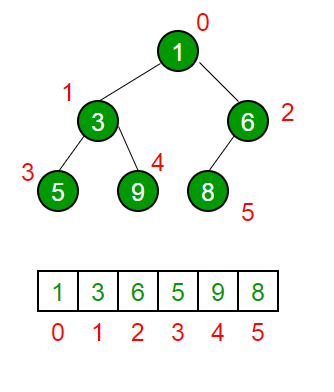
DATA STRUCTURES

HEAPS

Not to be mistaken with memory heap. A heap is an array-like structure that takes the form of a binary tree. Binary heaps are a common way of implementing *priority queues*; where each element has a priority.

Two types:

1. Min heap
2. Max heap

<< this array would look like:

[1, 3, 6, 5, 9, 8]

n = 6

*This is an example of a min heap*.

Math

1. A[n/2 + 1 …. n] are leaves. // items 4, 5, 6 are leaves above
2. Parent index – ((i - 1) / 2) // parent of 4 is (5-1) / 2 = 2 (index 2)
3. Left child 2i + 1 // 2 \* (index) 1 + 1 = 3
4. Right child 2i + 2 // 2 \* (index) 1 + 2 = 4

Operations

getMin() / getMax() - returns root element of min / max heap.

extractMin() / extractMax() - removes root element from heap

insert() - insert a key in to the heap

heapify() - recursively heapify a sub-tree to maintain shape

implementation of a min heap

class MinHeap {

private:

int size; // current size of heap

int capacity; // max size

vector<int> heap;

int parent(int i) { return (i – 1) / 2; }

int left(int i) { return 2 \* i + 1; }

int right(int i) { return 2 \* i + 2; }

public:

minHeap(capacity)

: capacity(capacity)

{

Size = 0;

heap.resize(capacity)

}

void insert(int k) {

if (size == capacity) { return; }

size++;

int i = size – 1;

heap[i] = k;

// fix the min heap property. Moves the element up until i >= parent or root

While (i != 0 && heap[parent(i)] > heap[i]) {

Swap(heap[i], heap[parent(i)];

i = parent(i);

}

}

Int extractMin() {

if (size == 0) { return -1; }

if (size == 1) {

size--;

return heap[0];

} else {

Int root = heap[0];

// maintain heap shape and then order

Heap[0] = heap[size – 1];

Size--;

Heapify(0);

Return root;

}

}

void heapify(int i) {

int l = left(i);

int r = right(i);

int smallest = i;

if ((l < size) && (heap[l] < heap[smallest]) {

smallest = l;

}

If ((r < size) && (heap[r] < heap[smallest]) {

Smallest = r;

}

// if the smallest is L or R, continue heapify

If (smallest != i) {

Swap(heap[i], heap[smallest]);

Heapify(smallest);

}

}

Void printHeap() {

Int power = 0;

Int value = 1;

For (int I = 0; I < size; i++) {

If (I == value) {

Std::cout << std::endl;

Power += 1;

Value += (I << power);

}

Std::cout << heap[i] << “ “;

}

Std::cout << std::endl;

}

}

Int n = 15;

minHeap heap(n);

srand(2);

for (int i = 0; i < n; i++) {

heap.insert(rand() % 100);

heap.printHeap();

}

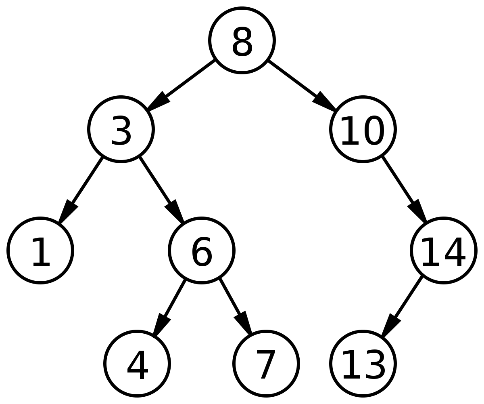
BINARY SEARCH TREE (BST)

A BST is different to a heap, which is an array, a BST is an actual tree with pointers;

1. To the parent
2. To the left child
3. To the right child

2 rules:

1. Smaller numbers than the parent go to the left child
2. Bigger numbers than the parent go to the right child



This means that to find the smallest number, just keep going left until you hit the leaf.

To find the biggest number, keep going right.

*h = height of tree*

* You can search a tree for a value in O(log n)
* You can compare < or > in O(1)
* You can insert in O(h)
  + Example to insert 15, the steps are:
    - Compare 15 with the root 8, it’s bigger, so move down the right
    - 15 is bigger than 10, move right again
    - 15 is bigger than 14, move right
    - There is no right, this is where we insert

Implementation

Struct node {

node\* left;

node\* right;

int data;

node(int d) {

data = d;

left = nullptr;

right = nullptr;

}

}

Class BST {

Public:

node\* root;

BST() { root = nullptr; }

node\* insertNode(node\* n, int d) {

if (n == nullptr) { // insertion point

return new node(d);

} else if (d < n->data) {

n->left = insertNode(n->left, d);

} else if (d > n->data) {

n->right = insertNode(n->right, d);

} else {

Assert(false); // duplicate?

}

Return n;

}

node\* search(int d) {

// tmp node to use for traversal

node\* tmp = root;

while(tmp != nullptr) {

if (d == tmp->data) {

return tmp;

} else if (d < tmp->data) {

tmp = tmp->left;

} else {

tmp = tmp->right;

}

}

// we didn’t find the node

Return nullptr;

}

}

Int data;

Vector<int> data\_vec;

Srand(2);

For (int i = 0; I < 10; i++) {

Data = rand() % 1000;

// save the number so we can search for it later

data\_vec.push\_back(data);

// insert node (save the root if it’s the first insert)

bst.root = bst.insertNode(bst.root, data);

}

for (int i : data\_vec) {

cout << “searching for node with data ” << i << endl;

bst.search(i);

}

DICTIONARY

A dictionary are key / value pairs. They are essentially an array where the indices are the key.

An implementation of a dictionary would be a…

HASH TABLE

A hash table is an implementation of a dictionary.

A hash is a function which is used to pick an index to store a value

Hash(x) = index;

For example:

Hash(name) = 3;

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|  |  |  | Ali |  |  |  |  |

Hash function

The hash function maps the Universe of keys into the slots of a hash table. A hash function can reduce the storage requirement to O(|K|) where k is the set of keys stored in a dictionary.

A hash function always produces the same output when the same input is used.

A collision is when a hash function produces the same key as another key which is used in a hash function. Using the example above; hash(randomKey) = 3… but index 3 already has a value!

A popular hash function is just:

Hash(int d) = d % m (table size)

There are a couple solutions for collision resolution:

1. Chaining
2. Open addressing

Chaining

Chaining provides a solution to collisions by having a linked list at each point, where, if the index is taken, the value is moved to the next item in the linked list until a spot is found. In the example above, *name* and *randomKey* both are stored under index 3… *name* will be the head of the linked list, *randomKey* will be the next item (tail). If another hash function returned with index 3, then this new item would be the tail, and *randomKey* would stay as the second item in the linked list.

When we search, we run the key through the same hash function, which produces the same index, so if we search for *randomKey*, this will go through hash(*randomKey*) to output index 3, we get to the head of the linked list, we move down until we find the key we are looking for.

Hashing also solves the issue of memory. Imagine a hash function which did the following:

Hash(x) = x2

Our array would have to be massive! Imagine we had index 8000, this hash function would output 64,000, which is essentially the position in the array to store our value!

Hashing reduces the Universal set into a smaller subset. The idea is to have *m =n* as close as possible, where *m = table size* and *n = number of items in table*.

Implementation

Class HashTable {

Private:

Int buckets; // number of buckets to store elements, e.g. length of array

List<int>\* table;

Public:

HashTable(int b) {

Buckets = b;

Table = new list<int>[buckets];

}

Void insert(int d) {

// simple hash function (modulo number of buckets)

Int bucket = d % buckets;

Table[bucket].push\_back(d);

}

Void print() {

For (int i = 0; I < buckets; i++) {

Cout << “| BUCKET “ << i << “ | “;

For (auto j : table[i]) {

Cout << “ -> | “ << j << “ | “;

}

Cout << endl;

}

}

}

HashTable ht(8);

Srand(2);

For (int i = 0; I < 20; i++) {

Ht.insert(rand() % 100);

}

ht.print();

operations

The basic dictionary operations; INSERT, SEARCH & DELETE, require only O(1) time on average. SEARCHING worst time is that of a linked list which is O(n), but in practice, this is rarely the case.

Open addressing

In open addressing, all elements occupy the hash table itself, that is each table entry contains either an element of the dynamic set, or NIL.

When searching for an element, we systematically examine table slots until either we find the desired element or we have ascertained that the element is not in the table.

With open addressing, the has table can ‘fill up’ so that no further insertions can be made. The advantage of open addressing is that it avoids pointers altogether.

The extra memory freed by not storing pointers provides the hash table with a larger number of slots for the same amount of memory, potentially yielding fewer collisions and retrieval.

To insert, we examine the hash table until we find an empty slot:

Instead of being fixed in the order 0, 1, …, m-1 (which requires O(n)), the sequence of positions probed depends upon the key being inserted.

Illustration

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 37 | 280 |  |  | 620 |  |  |  |

1. Insert 280 = hash(280, 1) = 1
2. Insert 37 = hash(37, 1) = 0
3. Insert 620 = hash(620, 1) = 1… oops, this is taken, try hash(620, 2) = 4
4. Search 620 = hash(620, 1)… hmm, this is not it, try hash(620, 2)… here it is!
5. Search 2 = hash(2, 1)… this slot is empty! It must not exist.
6. Delete 37 = hash(37, 1) = found. Replace with a *deleteMe* flag

Note when deleting we do not remove the value stored there, as this would screw up the searching. Instead we replace it with a *deleteMe* flag.. so when we search, if we come across a *deleteMe* flag, we keep searching.. we only stop when find what we are looking for, or we find an empty slot (which means the key we are searching does not exist).

When we insert, it is fine to overwrite a *deleteMe* flag.

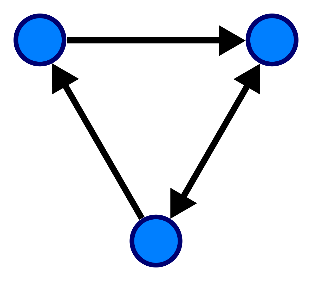
GRAPHS

A graph is a non-linear data structure consisting of nodes (AKA vertices) and edges.

v = {0, 1, 2, 3, 4} // v = vertices

Graphs can be:

1. Non-directional (like the image above)
2. Directional - You can only move in directions specified by the arrows

// directional graph

A

B

C

Adjacency list:

Adj[a] = {B}

Adj[b] = {C}

Adj[c] = {A, B}

A graph consists of the following components:

1. A finite set of vertices
2. A finite set of ordered pairs of the form (u, v)
   1. E.g. in directional example above (A, B) is valid, (B, A) is not
   2. Pairs include: {(A, B), (B, C), (C, A), (C, B)}

Adjacency List

An adjacency list is used to illustrate a graph.

Imagine the following:



An adjacency list would look something like:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | => | 1 | => | 4 |  |  |  |  |  |
| 1 | => | 0 | => | 4 | => | 2 | => | 3 |  |
| 2 | => | 1 | => | 3 |  |  |  |  |  |
| 3 | => | 1 | => | 4 | => | 2 |  |  |  |
| 4 | => | 3 | => | 0 | => | 1 |  |  |  |

For example:

in one move, vertex 0 can go to vertex 1 or 4

in one move, vertex 1 can go to vertex 0, 4, 2 or 3

etc…

Implementation

Class Graph {

Int v; // no. of vertices

List<int>\* adj;

Public:

Graph(int v)

: v(v)

{

Adj = new list<int>[v];

}

Void addEdge(int v, int w) {

Adj[v].push\_back(w);

}

Void printGraph() {

For (int I = 0; I < v; i++) {

Cout << “\nVertex “ << i << “\nhead”;

For (auto& x : adj[i]) {

Cout << “=> “ << x;

}

}

}

}

Breadth First Search (BFS)

BFS is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root, and explores all of the neighbour nodes at the present depth prior to moving on to the nodes at the next depth level.

It takes O(v + e) time; where v = vertices, and e =edges

It looks at nodes reachable in:

1. 0 moves {s}
2. 1 move adj[s]
3. 2 moves, etc…

You need to be careful to avoid duplicates, otherwise an infinite loop could occur.

V

C

Z

X

F

D

S

A

1. If *S* is our starting point (blue), we can reach it in 0 moves
2. We can reach A and X (orange) in one move
3. Z, D and C in two moves (green)
4. F and V in three moves

As we traverse through the graph, e.g from S => A, be careful not to include *S* at neighboring vertices at A as we have already included this at point 0. Doing so will cause an infinite loop where you just go between S => A and vice versa.

Implementation of BFS in our Graph class above

Add the following method to our Graph class:

Void Graph::BFS(int s) {

// mark all vertices as not visited

Bool \*visited = new bool[v];

For (int I = 0; I < v; i++) {

Visited[i] = false;

}

List<int> queue;

// mark current node as visited

Visited[s] = true;

Queue.push\_back(s);

List<int>::iterator i; // used to get all adjacent vertices of a vertex

While (!queue.empty()) {

// dequeue a vertex from a queue and print it

S = queue.front();

Cout << s << “ ”;

Queue.pop\_front();

// get all adjacent vertices of the dequeued vertex s

// if not previously visited, mark as visited and then enqueue it

For (int I = adj[s].begin(); I != adj[s].end(); ++i) {

If (!visited[\*i]) {

Visited[\*i] = true;

Queue.push\_back(\*i);

}

}

}

}

Int main() {

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

g.BFS(2); // 2 0 3 1

}

^^^ illustration of our graph

3

2

1

0

Depth First Search (DFS)

An algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node and explores as far as possible along each branch, before backtracking.