**Lab 6**

**Binomial Tree and Binomial Heaps**

1. **Make a Binomial Heap by Inserting a sequence of elements taken from the user.**

struct Node

{

int data, degree;

Node \*child, \*sibling, \*parent;

};

Node\* newNode(int key)

{

Node \*temp = new Node;

temp->data = key;

temp->degree = 0;

temp->child = temp->parent = temp->sibling = NULL;

return temp;

}

Node\* mergeBinomialTrees(Node \*b1, Node \*b2)

{

if (b1->data > b2->data)

swap(b1, b2);

b2->parent = b1;

b2->sibling = b1->child;

b1->child = b2;

b1->degree++;

return b1;

}

list<Node\*> unionBionomialHeap(list<Node\*> l1,

list<Node\*> l2)

{

list<Node\*> \_new;

list<Node\*>::iterator it = l1.begin();

list<Node\*>::iterator ot = l2.begin();

while (it!=l1.end() && ot!=l2.end())

{

if((\*it)->degree <= (\*ot)->degree)

{

\_new.push\_back(\*it);

it++;

}

else

{

\_new.push\_back(\*ot);

ot++;

}

}

while (it != l1.end())

{

\_new.push\_back(\*it);

it++;

}

while (ot!=l2.end())

{

\_new.push\_back(\*ot);

ot++;

}

return \_new;

}

list<Node\*> adjust(list<Node\*> \_heap)

{

if (\_heap.size() <= 1)

return \_heap;

list<Node\*> new\_heap;

list<Node\*>::iterator it1,it2,it3;

it1 = it2 = it3 = \_heap.begin();

if (\_heap.size() == 2)

{

it2 = it1;

it2++;

it3 = \_heap.end();

}

else

{

it2++;

it3=it2;

it3++;

}

while (it1 != \_heap.end())

{

if (it2 == \_heap.end())

it1++;

else if ((\*it1)->degree < (\*it2)->degree)

{

it1++;

it2++;

if(it3!=\_heap.end())

it3++;

}

else if (it3!=\_heap.end() &&

(\*it1)->degree == (\*it2)->degree &&

(\*it1)->degree == (\*it3)->degree)

{

it1++;

it2++;

it3++;

}

else if ((\*it1)->degree == (\*it2)->degree)

{

Node \*temp;

\*it1 = mergeBinomialTrees(\*it1,\*it2);

it2 = \_heap.erase(it2);

if(it3 != \_heap.end())

it3++;

}

}

return \_heap;

}

list<Node\*> insertATreeInHeap(list<Node\*> \_heap,

Node \*tree)

{

list<Node\*> temp;

temp.push\_back(tree);

temp = unionBionomialHeap(\_heap,temp);

return adjust(temp);

}

list<Node\*> removeMinFromTreeReturnBHeap(Node \*tree)

{

list<Node\*> heap;

Node \*temp = tree->child;

Node \*lo;

while (temp)

{

lo = temp;

temp = temp->sibling;

lo->sibling = NULL;

heap.push\_front(lo);

}

return heap;

}

list<Node\*> insert(list<Node\*> \_head, int key)

{

Node \*temp = newNode(key);

return insertATreeInHeap(\_head,temp);

}

Node\* getMin(list<Node\*> \_heap)

{

list<Node\*>::iterator it = \_heap.begin();

Node \*temp = \*it;

while (it != \_heap.end())

{

if ((\*it)->data < temp->data)

temp = \*it;

it++;

}

return temp;

}

list<Node\*> extractMin(list<Node\*> \_heap)

{

list<Node\*> new\_heap,lo;

Node \*temp;

temp = getMin(\_heap);

list<Node\*>::iterator it;

it = \_heap.begin();

while (it != \_heap.end())

{

if (\*it != temp)

{

new\_heap.push\_back(\*it);

}

it++;

}

lo = removeMinFromTreeReturnBHeap(temp);

new\_heap = unionBionomialHeap(new\_heap,lo);

new\_heap = adjust(new\_heap);

return new\_heap;

}

void printTree(Node \*h)

{

while (h)

{

cout << h->data << " ";

printTree(h->child);

h = h->sibling;

}

}

void printHeap(list<Node\*> \_heap)

{

list<Node\*> ::iterator it;

it = \_heap.begin();

while (it != \_heap.end())

{

printTree(\*it);

it++;

}

}

int main()

{

int ch,key;

list<Node\*> \_heap;

\_heap = insert(\_heap,10);

\_heap = insert(\_heap,20);

\_heap = insert(\_heap,30);

cout << "Heap elements after insertion:\n";

printHeap(\_heap);

Node \*temp = getMin(\_heap);

cout << "\nMinimum element of heap "

<< temp->data << "\n";

\_heap = extractMin(\_heap);

cout << "Heap after deletion of minimum element\n";

printHeap(\_heap);

return 0;

}

1. **Merge two binomial Heaps**

void merge(BinomialHeap h1) {

NodePtr curr1 = getHead();

NodePtr curr2 = h1.getHead();

NodePtr curr3 = nullptr;

NodePtr temp = nullptr;

if (curr1->degree <= curr2->degree) {

curr3 = curr1;

curr1 = curr1->sibling;

} else {

curr3 = curr2;

curr2 = curr2->sibling;

}

temp = curr3;

while(curr1 != nullptr && curr2 != nullptr) {

if (curr1->degree <= curr2->degree) {

curr3->sibling = curr1;

curr1 = curr1->sibling;

} else {

curr3->sibling = curr2;

curr2 = curr2->sibling;

}

curr3 = curr3->sibling;

}

if (curr1 != nullptr) {

while(curr1 != nullptr) {

curr3->sibling = curr1;

curr1 = curr1->sibling;

curr3 = curr3->sibling;

}

}

if (curr2 != nullptr) {

while(curr2 != nullptr) {

curr3->sibling = curr2;

curr2 = curr2->sibling;

curr3 = curr3->sibling;

}

}

curr3 = temp;

NodePtr prev = nullptr;

NodePtr next = curr3->sibling;

while (next != nullptr) {

if ((curr3->degree != next->degree )|| (next->sibling != nullptr && curr3->degree == next->sibling->degree)) {

prev = curr3;

curr3 = next;

} else {

if (curr3->data <= next->data) {

curr3->sibling = next->sibling;

BinomialHeap::linkBinomialTrees(curr3, next);

} else {

if (prev == nullptr) {

temp = next;

} else {

prev->sibling = next;

}

BinomialHeap::linkBinomialTrees(next, curr3);

curr3 = next;

}

}

next = curr3->sibling;

}

setHead(temp);

}

1. **Delete the smallest element in the Binomial Heap.**

NodePtr deleteMin() {

NodePtr curr = head;

NodePtr prevMin = nullptr;

NodePtr minPtr = nullptr;

NodePtr prevPtr = nullptr;

int min = 999999;

while (curr != nullptr) {

if (curr->data <= min) {

min = curr->data;

prevMin = prevPtr;

minPtr = curr;

}

prevPtr = curr;

curr = curr->sibling;

}

if (prevMin != nullptr && minPtr->sibling != nullptr) {

prevMin->sibling = minPtr->sibling;

} else if (prevMin != nullptr && minPtr->sibling == nullptr) {

prevMin->sibling = nullptr;

}else if(prevMin == nullptr and minPtr->sibling != nullptr) {

head = minPtr->sibling;

}else if(prevMin == nullptr and minPtr->sibling == nullptr) {

head = nullptr;

}

NodePtr childPtr = minPtr->child;

while (childPtr != nullptr) {

childPtr->p = nullptr;

childPtr = childPtr->sibling;

}

stack<NodePtr> s;

childPtr = minPtr->child;

while (childPtr != nullptr) {

s.push(childPtr);

childPtr = childPtr->sibling;

}

curr = s.top();

NodePtr temp = curr;

s.pop();

while (!s.empty()) {

curr->sibling = s.top();

s.pop();

curr = curr->sibling;

}

curr->sibling = nullptr;

BinomialHeap h;

h.setHead(temp);

merge(h);

return minPtr;}

**Lab-7**

**Hashing with collision resolution**

**1**.import random

import time

class HashTable:

def \_\_init\_\_(self, size):

self.size = size

self.table = [[] for \_ in range(size)]

def \_hash\_function(self, key):

return key % self.size

def insert(self, key):

hash\_value = self.\_hash\_function(key)

bucket = self.table[hash\_value]

for i, k in enumerate(bucket):

if k == key:

return

bucket.append(key)

def search(self, key):

hash\_value = self.\_hash\_function(key)

bucket = self.table[hash\_value]

for k in bucket:

if k == key:

return True

return False

def delete(self, key):

hash\_value = self.\_hash\_function(key)

bucket = self.table[hash\_value]

for i, k in enumerate(bucket):

if k == key:

del bucket[i]

return True

return False

# Create a hash table with separate chaining that can hold up to 10,000 integers.

hash\_table = HashTable(10000)

# Generate 10,000 random integers and insert them into the hash table.

for i in range(10000):

hash\_table.insert(random.randint(1, 100000))

# Print out the contents of the hash table.

print(hash\_table.table)

# Calculate the load factor of the hash table.

load\_factor = sum(len(bucket) for bucket in hash\_table.table) / len(hash\_table.table)

print(f"Load factor: {load\_factor}")

# Measure the time it takes to search for a random integer in the hash table.

start\_time = time.time()

hash\_table.search(random.randint(1, 100000))

end\_time = time.time()

print(f"Search time: {end\_time - start\_time} seconds")

# Measure the time it takes to remove a random integer from the hash table.

start\_time = time.time()

hash\_table.delete(random.randint(1, 100000))

end\_time = time.time()

print(f"Delete time: {end\_time - start\_time} seconds")

# Repeat steps 5 and 6 for different load factors (e.g. 0.5, 0.75, 0.9) and record the results.

load\_factors = [0.5, 0.75, 0.9]

for lf in load\_factors:

# Create a new hash table with the desired load factor.

hash\_table = HashTable(int(10000 \* lf))

# Insert random integers into the hash table until the desired load factor is reached.

while sum(len(bucket) for bucket in hash\_table.table) / len(hash\_table.table) < lf:

hash\_table.insert(random.randint(1, 100000))

# Measure the time it takes to search for a random integer in the hash table.

start\_time = time.time()

hash\_table.search(random.randint(1, 100000))

end\_time = time.time()

print(f"Load factor {lf}: Search time: {end\_time - start\_time} seconds")

# Measure the time it takes to remove a random integer from the hash table.

start\_time = time.time()

hash\_table.delete(random.randint(1, 100000))

end\_time = time.time()

print(f"Load factor {lf}: Delete time: {end\_time - start\_time} seconds")

**2**.import random

import time

class HashTable:

def \_\_init\_\_(self, size):

self.size = size

self.table = [None] \* size

def \_hash\_function(self, key):

return key % self.size

def insert(self, key):

index = self.\_hash\_function(key)

for i in range(self.size):

if self.table[index] is None:

self.table[index] = key

return

if self.table[index] == key:

return

index = (index + 1) % self.size

def search(self, key):

index = self.\_hash\_function(key)

for i in range(self.size):

if self.table[index] == key:

return True

if self.table[index] is None:

return False

index = (index + 1) % self.size

return False

# Create a hash table with linear probing that can hold up to 10,000 integers.

hash\_table = HashTable(10000)

# Generate 5,000 random integers and insert them into the hash table.

for i in range(5000):

hash\_table.insert(random.randint(1, 100000))

# Print out the contents of the hash table.

print(hash\_table.table)

# Calculate the load factor of the hash table.

load\_factor = sum(1 for x in hash\_table.table if x is not None) / len(hash\_table.table)

print(f"Load factor: {load\_factor}")

# Measure the time it takes to search for a random integer in the hash table.

start\_time = time.time()

hash\_table.search(random.randint(1, 100000))

end\_time = time.time()

print(f"Search time: {end\_time - start\_time} seconds")

# Measure the time it takes to insert a random integer into the hash table.

start\_time = time.time()

hash\_table.insert(random.randint(1, 100000))

end\_time = time.time()

print(f"Insert time: {end\_time - start\_time} seconds")

# Repeat steps 5 and 6 for different load factors (e.g. 0.5, 0.75, 0.9) and record the results.

load\_factors = [0.5, 0.75, 0.9]

for lf in load\_factors:

# Create a new hash table with the desired load factor.

hash\_table = HashTable(int(10000 \* lf))

# Insert random integers into the hash table until the desired load factor is reached.

while sum(1 for x in hash\_table.table if x is not None) / len(hash\_table.table) < lf:

hash\_table.insert(random.randint(1, 100000))

# Measure the time it takes to search for a random integer in the hash table.

start\_time = time.time()

hash\_table.search(random.randint(1, 100000))

end\_time = time.time()

print(f"Load factor {lf}: Search time: {end\_time - start\_time} seconds")

# Measure the time it takes to insert a random integer into the hash table.

start\_time = time.time()

hash\_table.insert(random.randint(1, 100000))

end\_time = time.time()

print(f"Load factor {lf}: Insert time: {end\_time - start\_time} seconds")

**3**.import random

import time

class HashTable:

def \_\_init\_\_(self, size):

self.size = size

self.table = [None] \* size

def \_hash\_function1(self, key):

return key % self.size

def \_hash\_function2(self, key):

return 7 - (key % 7)

def insert(self, key):

index = self.\_hash\_function1(key)

offset = self.\_hash\_function2(key)

for i in range(self.size):

if self.table[index] is None:

self.table[index] = key

return

if self.table[index] == key:

return

index = (index + offset) % self.size

def search(self, key):

index = self.\_hash\_function1(key)

offset = self.\_hash\_function2(key)

for i in range(self.size):

if self.table[index] == key:

return True

if self.table[index] is None:

return False

index = (index + offset) % self.size

return False

def remove(self, key):

index = self.\_hash\_function1(key)

offset = self.\_hash\_function2(key)

for i in range(self.size):

if self.table[index] == key:

self.table[index] = None

return

if self.table[index] is None:

return

index = (index + offset) % self.size

# Create a hash table with double hashing that can hold up to 10,000 integers.

hash\_table = HashTable(10000)

# Generate 7,000 random integers and insert them into the hash table.

for i in range(7000):

hash\_table.insert(random.randint(1, 100000))

# Print out the contents of the hash table.

print(hash\_table.table)

# Calculate the load factor of the hash table.

load\_factor = sum(1 for x in hash\_table.table if x is not None) / len(hash\_table.table)

print(f"Load factor: {load\_factor}")

# Measure the time it takes to search for a random integer in the hash table.

start\_time = time.time()

hash\_table.search(random.randint(1, 100000))

end\_time = time.time()

print(f"Search time: {end\_time - start\_time} seconds")

# Measure the time it takes to remove a random integer from the hash table.

start\_time = time.time()

hash\_table.remove(random.randint(1, 100000))

end\_time = time.time()

print(f"Remove time: {end\_time - start\_time} seconds")

# Repeat steps 5 and 6 for different load factors (e.g. 0.5, 0.75, 0.9) and record the results.

load\_factors = [0.5, 0.75, 0.9]

for lf in load\_factors:

# Create a new hash table with the desired load factor.

hash\_table = HashTable(int(10000 \* lf))

# Insert random integers into the hash table until the desired load factor is reached.

while sum(1 for x in hash\_table.table if x is not None) / len(hash\_table.table) < lf:

hash\_table.insert(random.randint(1, 100000))

# Measure the time it takes to search for a random integer in the hash table.

start\_time = time.time()

hash\_table.search(random.randint(1, 100000))

end\_time = time.time()

print(f"Load factor {lf}: Search time: {end\_time - start\_time} seconds")

# Measure

**4**.import random

import time

class QuadraticProbingHashTable:

def \_\_init\_\_(self, size):

self.size = size

self.keys = [None] \* self.size

self.values = [None] \* self.size

self.count = 0

def hash\_function(self, key):

return key % self.size

def insert(self, key, value):

if self.count == self.size:

print("Hash table is full.")

return False

index = self.hash\_function(key)

if self.keys[index] is None:

self.keys[index] = key

self.values[index] = value

self.count += 1

return True

i = 1

while i < self.size:

new\_index = (index + i \*\* 2) % self.size

if self.keys[new\_index] is None:

self.keys[new\_index] = key

self.values[new\_index] = value

self.count += 1

return True

i += 1

print("Could not insert key-value pair.")

return False

def search(self, key):

index = self.hash\_function(key)

if self.keys[index] == key:

return self.values[index]

i = 1

while self.keys[(index + i \*\* 2) % self.size] != key:

i += 1

return self.values[(index + i \*\* 2) % self.size]

def test\_quadratic\_probing():

table = QuadraticProbingHashTable(10000)

nums = random.sample(range(1000000), 2000)

# Insert 2000 random numbers

start\_time = time.time()

for num in nums:

table.insert(num, num)

end\_time = time.time()

print("Time taken to insert 2000 numbers:", end\_time - start\_time)

# Print table contents

print("Table contents:")

for i in range(len(table.keys)):

if table.keys[i] is not None:

print(table.keys[i], table.values[i])

# Calculate load factor

load\_factor = table.count / table.size

print("Load factor:", load\_factor)

# Search for a random number

start\_time = time.time()

random\_num = random.choice(nums)

result = table.search(random\_num)

end\_time = time.time()

print("Time taken to search for a random number:", end\_time - start\_time)

# Insert a random number

start\_time = time.time()

new\_num = random.randint(1000000, 2000000)

table.insert(new\_num, new\_num)

end\_time = time.time()

print("Time taken to insert a new number:", end\_time - start\_time)

test\_quadratic\_probing()

**5**.import random

import time

# Hash table with separate chaining

class HashTableSeparateChaining:

def \_\_init\_\_(self, capacity):

self.capacity = capacity

self.size = 0

self.table = [[] for \_ in range(capacity)]

def \_\_len\_\_(self):

return self.size

def \_hash(self, key):

return key % self.capacity

def insert(self, key):

hash\_val = self.\_hash(key)

bucket = self.table[hash\_val]

for i, (k, v) in enumerate(bucket):

if k == key:

bucket[i] = (k, v+1)

break

else:

bucket.append((key, 1))

self.size += 1

def search(self, key):

hash\_val = self.\_hash(key)

bucket = self.table[hash\_val]

for k, v in bucket:

if k == key:

return v

return None

def delete(self, key):

hash\_val = self.\_hash(key)

bucket = self.table[hash\_val]

for i, (k, v) in enumerate(bucket):

if k == key:

del bucket[i]

self.size -= 1

return

raise KeyError(f"Key {key} not found in table")

# Hash table with linear probing

class HashTableLinearProbing:

def \_\_init\_\_(self, capacity):

self.capacity = capacity

self.size = 0

self.keys = [None] \* capacity

self.values = [None] \* capacity

def \_\_len\_\_(self):

return self.size

def \_hash(self, key):

return key % self.capacity

def \_probe(self, hash\_val):

i = 0

while self.keys[(hash\_val + i) % self.capacity] is not None:

i += 1

return (hash\_val + i) % self.capacity

def insert(self, key, value):

hash\_val = self.\_hash(key)

if self.keys[hash\_val] is None:

self.keys[hash\_val] = key

self.values[hash\_val] = value

self.size += 1

else:

i = self.\_probe(hash\_val)

self.keys[i] = key

self.values[i] = value

self.size += 1

def search(self, key):

hash\_val = self.\_hash(key)

i = 0

while self.keys[(hash\_val + i) % self.capacity] is not None:

if self.keys[(hash\_val + i) % self.capacity] == key:

return self.values[(hash\_val + i) % self.capacity]

i += 1

return None

def delete(self, key):

hash\_val = self.\_hash(key)

i = 0

while self.keys[(hash\_val + i) % self.capacity] is not None:

if self.keys[(hash\_val + i) % self.capacity] == key:

self.keys[(hash\_val + i) % self.capacity] = None

self.values[(hash\_val + i) % self.capacity] = None

self.size -= 1

return

i += 1

raise KeyError(f"Key {key} not found in table")

# Hash table with double hashing

class HashTableDoubleHashing:

def \_\_init\_\_(self, capacity):

self.capacity = capacity

self.size = 0

self.keys = [None] \* capacity

self.values = [None