Algo HW05

Stav Rones

3/27/2020

1. Hash Tables

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Key Values | Probe Sequence |  | index | Final Hash Table Contents |
| 43 | 0 |  | 0 | 43 |
| 23 | 6 |  | 1 | 0 |
| 1 | 3 |  | 2 | 31 |
| 0 | 1 |  | 3 | 1 |
| 15 | 7 |  | 4 | 5 |
| 31 | 2 |  | 5 | 7 |
| 4 | 9 |  | 6 | 23 |
| 7 | 5 |  | 7 | 15 |
| 11 | 5, 6, 7, 8 |  | 8 | 11 |
| 3 | 7, 8, 9, 10 |  | 9 | 4 |
| 5 | 0, 1, 2, 3, 4 |  | 10 | 3 |
| 9 | 10, 0, 1, 2, 3, 4, 5, 6, 7 ,8 ,9  (Not inserted) |  |

1. Heaps

Proof by induction: Left Child

1. Left child of root (items[0]) ?= items[2\*0+1] = items[1] **True**
2. Assume left child of items[k] = items[2k+1]
3. Prove left child of items[k+1] = items[2(k+1)+1]:

Since the left child of items[k] is items[2k+1], this means that the right child is at element [2K+2] and the left child of the next element (items[k+1]) is at items[2k+3] = items[2(k+1)+1]

Proof by induction: Left Child

1. Left child of root (items[0]) ?= items[2\*0+1] = items[1] **True**
2. Assume left child of items[k] = items[2k+1]
3. Prove left child of items[k+1] = items[2(k+1)+1:

Since the left child of items[k] is items[2k+1], this means that the right child is at element [2K+2] and the left child of the next element (items[k+1]) is at items[2k+3] = items[2(k+1)+1]

Proof by induction: Parent

1. Parent of items[1] ?= items[] = items[0]
2. Assume parent of child items[k] = items[]
3. Prove parent of items[k+1] = items[] = items[]

We know that if the root is indexed at i=0, then left children will always have odd indices and right children will always have even indices. This means that if k = odd, then the parent of items[k+1] will be equal to the parent of items[k], so:

When Parent(k) = parent(k+1), does = ?

Since (k = 2n+1) ->

= ->

= ->

= ->

n = n **True**

If k = even, then the parent of items[k+1] should be the parent(k) element + 1, so:

When parent(k+1) = parent(k)+1, does = +1?

Since (k=2n) ->

= ->

= ->

n = (n-1)+1 = n **True**

3. Graphs

3. BFS traversal queue 4. DFS traversal

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Node visited | Queue (front to back) |  | Node visited | Stack (bottom to top) |
| a | a |  | a | a |
| b | b |  | b | a,b |
| c | b,c |  | c | a,b,c |
| d | b,c,d |  | d | a,b,c,d |
| - | c,d |  | f | a,b,c,d,f |
| e | d,e |  | e | a,b,c,d,f,e |
| f | e,f |  | g | a,b,c,d,f,e,g |
| h | e,f,h |  | - | a,b,c,d,f,e |
| g | h,f,g |  | i | a,b,c,d,f,e,i |
| i | h,f,g,i |  | - | a,b,c,d,f,e |
| - | f,g,i |  | - | a,b,c,d,f |
| - | g,i |  | - | a,b,c,d |
| - | i |  | h | a,b,c,d,h |
| - | - |  | - | a,b,c,d |
|  |  |  | - | a,b,c |
|  |  |  | - | a,b |
|  |  |  | - | a |
|  |  |  | - | - |

5. A) Prims Algorithm: Tree sum = 21

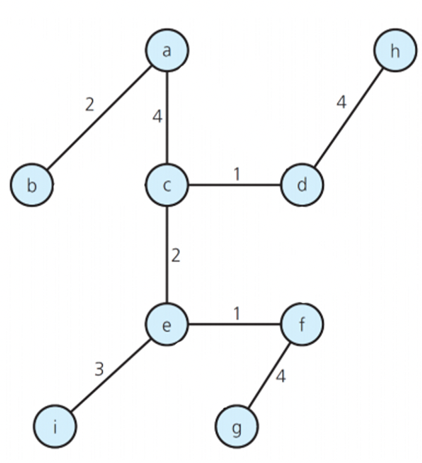
|  |  |  |
| --- | --- | --- |
| Nodes Visited | Edges used | Current Edge Choices / Weights |
| A | - | AB(2),AC(4),AD(6) |
| B | AB(2) | AC(4),AD(6),BC(5) |
| C | AC(4) | AD(6), CD(1),CE(2) |
| D | CD(1) | CE(2),DF(3),DH(4) |
| E | CE(2) | DF(3),DH(4),EI(3),EG(5),EF(1) |
| F | EF(1) | DH(4),EI(3),EG(5),FG(4) |
| I | EI(3) | DH(4), EG(5),FG(4) |
| H | DH(4) | EG(5),FG(4) |
| G | FG(4) | - |

B) Kruskal’s Algorithm: Tree sum =

|  |  |
| --- | --- |
| Edges | Add (Y/N) |
| CD (1) | Y |
| EF(1) | Y |
| CE(2) | Y |
| AB(2) | Y |
| DF(3) | N |
| EI(3) | Y |
| GF(4) | Y |
| AC(4) | Y |
| DH(4) | Y |
| BC(5) | N |
| EG(5) | N |
| AD(6) | N |

Minimum Spanning Tree for Prim’s and Kruskal’s algorithms:

Delete edges: [AD,BC,DF, EG]

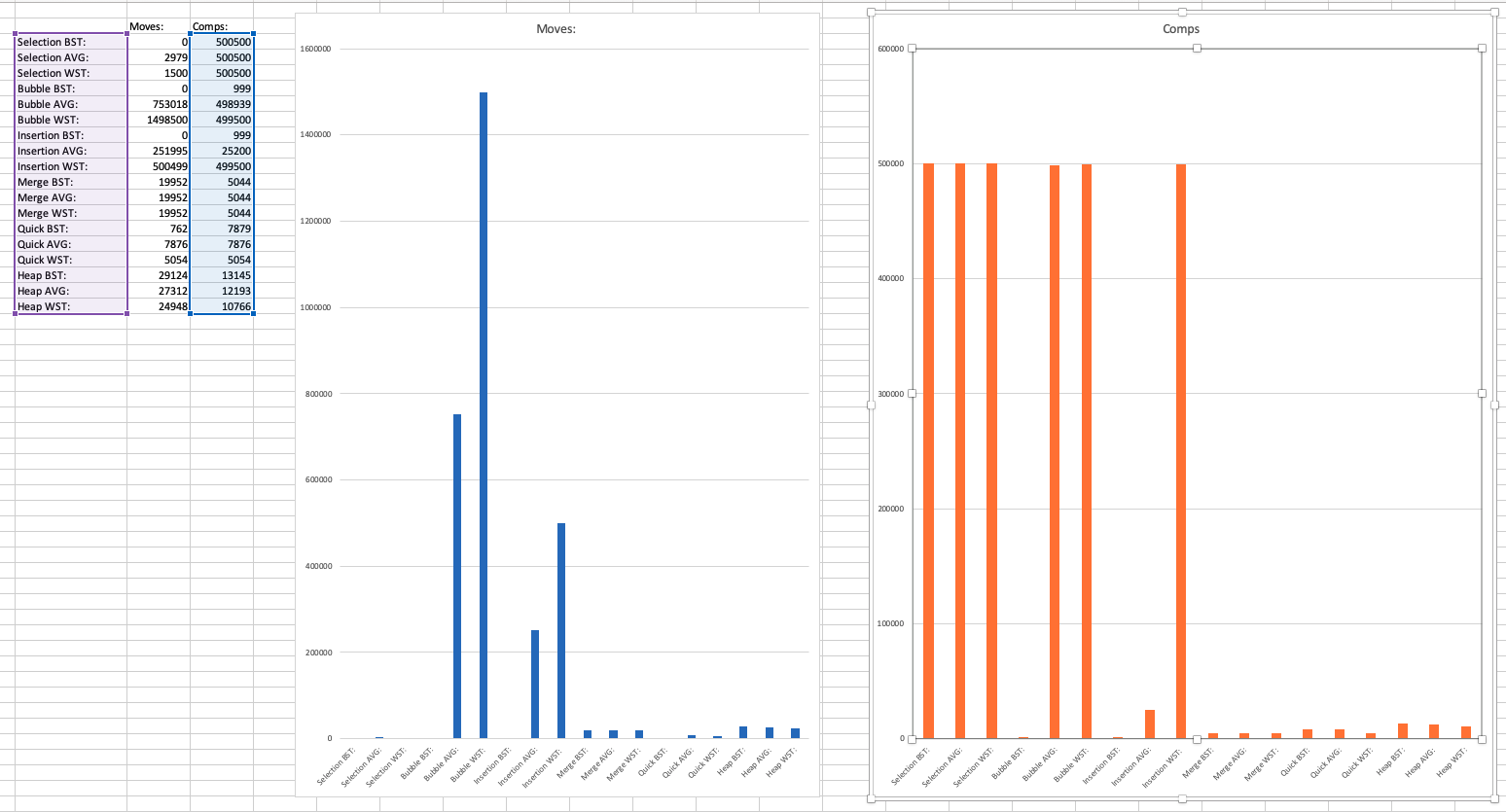


**Extra Credit 1:**

C++ Source files:

EC1.cpp

sort.h



Heap sort has similar moves to Merge sort, but slightly more comparisons for merge sort in all cases. Unlike the other algorithms, heap took the most comparisons and moves for the best case, and the least for the worst case. This has to do with the fact creating a heap from the array is easier for the “worst” case since it more similarly represents a heap.

**EC2:**

C++ Source files:

HW04.2\_main.cpp

bank.h

input.txt

output.txt

Modified the PQueue class to include private functions:

bool heapInsert(Event ev);

bool heapDelete();

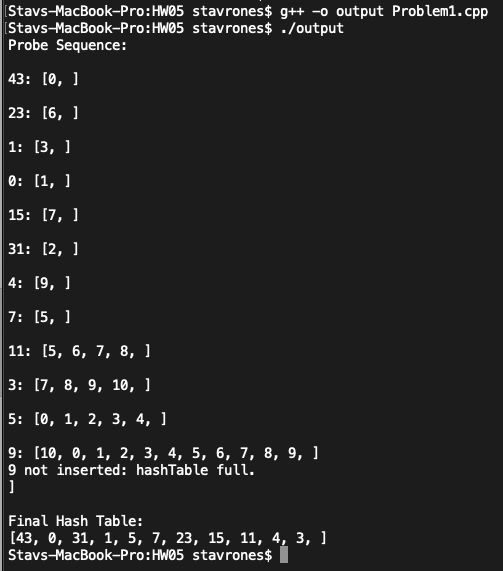
Event heapPeek();

void heapRebuild(int rootIndex);

PQueue implemented the Event events[] to be a minheap. The functions then inserted and deleted nodes of the heap according to ADTime and EventType. heapPeek returns the root of the heap

**Summary to approach to solving programming problems:**

For problem 1, I implemented the hash function and hash table in c++ in order to use print statements to output the desired answers.



For EC1, I simply repeated the process used for HW4 to implement the algorithm, mark comparisons and moves, and use the output text data file to create the excel graph.

For EC2, I changed the public PQueue insert, delete, and peek functions but changed them to call heapInsert, heapDelete, and heapPeek instead of their list equivalents. For implementing an array heap, the peek method simply returned the first element, which is the root of the minheap. For insert and delete, the methods from lecture were built upon with the added comparison of EventType on top of ADTime. This was certainly more challenging than implementing a simple heap with one priority characteristic per node.

**Summary of the skills I acquired and challenges I faced by implementing the programs:**

By programming heap sort and different heap functionalities, it allowed me to get a hands on experience with the heap data structure, and solidified my knowledge of exactly how this data structure is achieved and implemented. Additionally, programming a hash function and table allowed me to achieve a better understanding of how hash tables and functions are implemented.

**My recommendations of extension to the programming problems:**

My only recommendation would be to have programming the hash function and table be required for the homework grade, and showing the output of the program instead of filling out the table by hand.