HANYANG UNIVERSITY
DIVISION OF COMPUTER SCIENCE

Data Structures

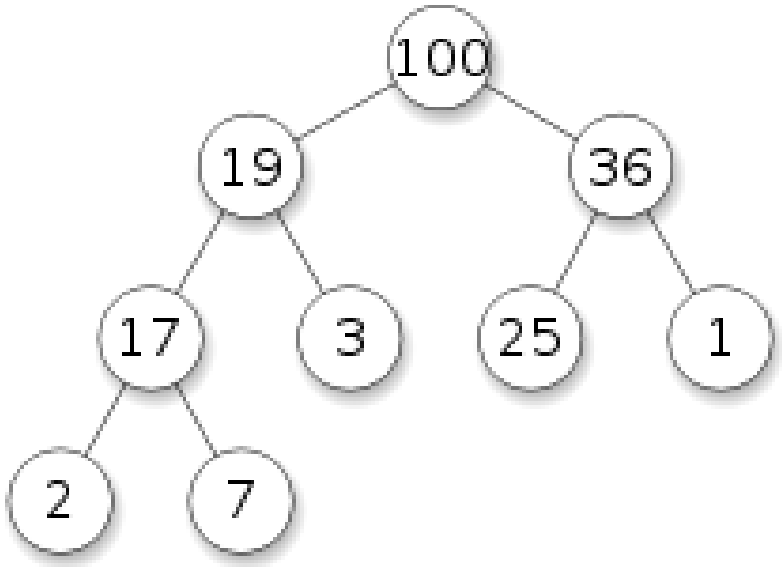
박영준 교수님

Lab8: Heap

Heap

- Specialized tree-based data structure which is essentially an almost complete tree that satisfies the heap property
- Min heap : for any given node C, if P is parent node of C, then the key of P is less than or equal to the key of C
- Max heap : for any given node C, if P is parent node of C, then the key of P is greater than or equal to the key of C

Heap



Heap using Array, Represented as Binary Tree

- Use the index of the array as the Node ID
- Elements of array are structure variables
  - typedef struct {
    int Priority;
    DATATYPE Data;
    } Element;
  - typedef struct {
    Element arr[];
    int NumofData;
    } Heap;

Simple Heap ADT

- void HeapInit(Heap *THeap);
  - Initialize heap → set NumofData to 0
- int IsEmpty(Heap *THeap);
  - Check heap is empty
  - If empty, return 1
  - else, return 0

Simple Heap ADT

- void HeapInsert(Heap *THeap, DATATYPE Data, int Priority);
  - Insert data with priority into heap
  - Compare priority with saved Elements
  - When i is the Node ID,
    Parent of Node : i/2
    Left child of Node : 2 * i
    Right child of Node : 2 * i + 1

Simple Heap ADT

- DATATYPE HeapDelete(Heap *THeap);
  - Delete node has highest priority
  - Replace the root with the last node
  - Reconstruct heap

Simple Heap ADT

- int RetIndexParent(int index);
  - Return index of parent
- int RetIndexLeftChild(int index);
  - Return index of left child
- int RetIndexRightChild(int index);
  - Return index of right child

Simple Heap ADT

- int RetIndexHighterPriorityOfChild(Heap *THeap, int index);
  - Return child node of index which has higher priority
  - If there are no childs, return 0
  - If there are only one child exists, return index of it

Simple Heap

```c
#include <stdio.h>

#define TRUE 1
#define FALSE 0

#define HEAP_LEN 100

typedef char DATATYPE;

typedef struct
{
    int Priority; //lowest value is highest priority
    DATATYPE Data;
} Element;

typedef struct Heap
{
    Element arr[HEAP_LEN];
    int NumofData;
} Heap;

void HeapInit(Heap *THeap);
int IsEmpty(Heap *THeap);

void HeapInsert(Heap *THeap, DATATYPE Data, int Priority);
DATATYPE HeapDelete(Heap *THeap);

int RetIndexParent(int index);
int RetIndexLeftChild(int index);
int RetIndexRightChild(int index);

int RetIndexHigherPriorityOfChilds(Heap *THeap, int index);

void PrintAll(Heap *THeap);
```

```c
void HeapInit(Heap *THeap)
{
    THeap->NumofData = 0;
}

int IsEmpty(Heap *THeap)
{
    if(THeap->NumofData == 0)
    {
        return TRUE;
    }
    else
    {
        return FALSE;
    }
}

void HeapInsert(Heap *THeap, DATATYPE Data, int Priority)
{
    int index = THeap->NumofData + 1;
    Element temp = {Priority, Data};

    while(index != 1)
    {
        if(Priority < (THeap->arr[RetIndexParent(index)].Priority))
        {
            THeap->arr[index] = THeap->arr[RetIndexParent(index)];
            index = RetIndexParent(index);
        }
        else
        {
            break;
        }
    }

    THeap->arr[index] = temp;
    THeap->NumofData += 1;
}
```

Simple Heap

```
108 DATATYPE HeapDelete(Heap *THeap)
109 {
110     DATATYPE temp = (THeap->arr[1]).Data;
111     Element last = THeap->arr[THeap->NumofData];
112
113     int ParentIndex = 1;
114     int ChildIndex;
115
116     while(ChildIndex = RetIndexHigherPriorityOfChilds(THeap, ParentIndex))
117     {
118         if(last.Priority <= THeap->arr[ChildIndex].Priority)
119         {
120             break;
121         }
122         THeap->arr[ParentIndex] = THeap->arr[ChildIndex];
123         ParentIndex = ChildIndex;
124     }
125
126     THeap->arr[ParentIndex] = last;
127     THeap->NumofData -= 1;
128     return temp;
129 }
130
131 int RetIndexParent(int index)
132 {
133     return index / 2;
134 }
135
136 int RetIndexLeftChild(int index)
137 {
138     return index * 2;
139 }
140
141 int RetIndexRightChild(int index)
142 {
143     return RetIndexLeftChild(index) + 1;
144 }
```

```
146 int RetIndexHigherPriorityOfChilds(Heap *THeap, int index)
147 {
148     if(RetIndexLeftChild(index) > THeap->NumofData)
149     {
150         return 0;
151     }
152     else if(RetIndexLeftChild(index) == THeap->NumofData)
153     {
154         return RetIndexLeftChild(index);
155     }
156     else
157     {
158         if(THeap->arr[RetIndexLeftChild(index)].Priority > THeap->arr[RetIndexRightChild(index)].Priority)
159         {
160             return RetIndexRightChild(index);
161         }
162         else
163         {
164             return RetIndexLeftChild(index);
165         }
166     }
167 }
168
169 void PrintAll(Heap *THeap)
170 {
171     for(int i = 1; i < THeap->NumofData + 1; i++)
172     {
173         printf("%d : %d %c\n", i, THeap->arr[i].Priority, THeap->arr[i].Data);
174     }
175 }
```

Simple Heap

```
36 int main(int argc, char *argv[])
37 {
38     Heap heap;
39     HeapInit(&heap);
40
41     HeapInsert(&heap, 'T', 1);
42     HeapInsert(&heap, 'H', 2);
43     HeapInsert(&heap, 'I', 3);
44     HeapInsert(&heap, 'S', 4);
45     HeapInsert(&heap, 'I', 5);
46     HeapInsert(&heap, 'S', 6);
47     HeapInsert(&heap, 'H', 7);
48     HeapInsert(&heap, 'E', 8);
49     HeapInsert(&heap, 'A', 9);
50     HeapInsert(&heap, 'P', 10);
51
52     PrintAll(&heap);
53     printf("\n");
54
55     printf("%c\n", HeapDelete(&heap));
56     printf("\n");
57
58     PrintAll(&heap);
59     printf("\n");
60
61     while(!IsEmpty(&heap))
62     {
63         printf("%c\n", HeapDelete(&heap));
64     }
65
66     return 0;
67 }
```

Heap

- Prioritized automatically based on data.
- ex) In min heap, when the data is character
  - ‘A’ has highest priority
  - ‘Z’ has lowest priority

Heap ADT

- void HeapInit(Heap *THeap);
  - Initialize heap → set NumofData to 0
- int IsEmpty(Heap *THeap);
  - Check heap is empty
  - If empty, return 1
  - else, return 0

Heap ADT

- void HeapInsert(Heap *THeap, DATATYPE Data);
  - Insert data considering its priority
  - When i is the Node ID,
    Parent of Node : i/2
    Left child of Node : 2 * i
    Right child of Node : 2 * i + 1

Heap ADT

- DATATYPE HeapDelete(Heap *THeap);
  - Delete node has highest priority
  - Replace the root with the last node
  - Reconstruct heap

Heap ADT

- int RetIndexParent(int index);
  - Return index of parent
- int RetIndexLeftChild(int index);
  - Return index of left child
- int RetIndexRightChild(int index);
  - Return index of right child

Heap ADT

- int RetIndexHighterPriorityOfChild(Heap *THeap, int index);
  - Return child node of index which has higher priority
  - If there are no childs, return 0
  - If there are only one child exists, return index of it

Min Heap

```c
#include <stdio.h>

#define TRUE 1
#define FALSE 0

#define HEAP_LEN 100

typedef char DATATYPE;

typedef struct Heap
{
    int NumofData;
    DATATYPE arr[HEAP_LEN];
} Heap;

void HeapInit(Heap *THeap);
int IsEmpty(Heap *THeap);

void HeapInsert(Heap *THeap, DATATYPE Data);
DATATYPE HeapDelete(Heap *THeap);

int RetIndexParent(int index);
int RetIndexLeftChild(int index);
int RetIndexRightChild(int index);

int RetIndexHigherPriorityOfChilds(Heap *THeap, int index);

void PrintAll(Heap *THeap);
```

```c
void HeapInit(Heap *THeap)
{
    THeap->NumofData = 0;
}

int IsEmpty(Heap *THeap)
{
    if(THeap->NumofData == 0)
    {
        return TRUE;
    }
    else
    {
        return FALSE;
    }
}

void HeapInsert(Heap *THeap, DATATYPE Data)
{
    int index = THeap->NumofData + 1;

    while(index != 1)
    {
        if((Data - THeap->arr[RetIndexParent(index)]) > 0)
        {
            THeap->arr[index] = THeap->arr[RetIndexParent(index)];
            index = RetIndexParent(index);
        }
        else
        {
            break;
        }
    }

    THeap->arr[index] = Data;
    THeap->NumofData +=1;
}
```

Min Heap

```
101 DATATYPE HeapDelete(Heap *THeap)
102 {
103     DATATYPE Data = THeap->arr[1];
104     DATATYPE last = THeap->arr[THeap->NumofData];
105 
106     int ParentIndex = 1;
107     int ChildIndex;
108 
109     while(ChildIndex = RetIndexHigherPriorityOfChilds(THeap, ParentIndex))
110     {
111         if((THeap->arr[ChildIndex] - last) >= 0)
112         {
113             break;
114         }
115 
116         THeap->arr[ParentIndex] = THeap->arr[ChildIndex];
117         ParentIndex = ChildIndex;
118     }
119 
120     THeap->arr[ParentIndex] = last;
121     THeap->NumofData -= 1;
122     return Data;
123 }
124 
125 int RetIndexParent(int index)
126 {
127     return index / 2;
128 }
129 
130 int RetIndexLeftChild(int index)
131 {
132     return index * 2;
133 }
134 
135 int RetIndexRightChild(int index)
136 {
137     return RetIndexLeftChild(index) + 1;
138 }
```

```
140 int RetIndexHigherPriorityOfChilds(Heap *THeap, int index)
141 {
142     if(RetIndexLeftChild(index) > THeap->NumofData)
143     {
144         return 0;
145     }
146     else if(RetIndexLeftChild(index) == THeap->NumofData)
147     {
148         return RetIndexLeftChild(index);
149     }
150     else
151     {
152         if((THeap->arr[RetIndexRightChild(index)] - THeap->arr[RetIndexLeftChild(index)]) < 0)
153         {
154             return RetIndexRightChild(index);
155         }
156         else
157         {
158             return RetIndexLeftChild(index);
159         }
160     }
161 }
162 
163 void PrintAll(Heap *THeap)
164 {
165     for(int i = 1; i < THeap->NumofData + 1; i++)
166     {
167         printf("%d : %c\n", i, THeap->arr[i]);
168     }
169 }
```

Min Heap

```
30 int main(int argc, char *argv[])
31 {
32     Heap heap;
33     HeapInit(&heap);
34
35     HeapInsert(&heap, 'T');
36     HeapInsert(&heap, 'H');
37     HeapInsert(&heap, 'I');
38     HeapInsert(&heap, 'S');
39     HeapInsert(&heap, 'I');
40     HeapInsert(&heap, 'S');
41     HeapInsert(&heap, 'H');
42     HeapInsert(&heap, 'E');
43     HeapInsert(&heap, 'A');
44     HeapInsert(&heap, 'P');
45
46     PrintAll(&heap);
47     printf("\n");
48
49     printf("%c\n", HeapDelete(&heap));
50     printf("\n");
51
52     PrintAll(&heap);
53     printf("\n");
54
55     while(!IsEmpty(&heap))
56     {
57         printf("%c\n", HeapDelete(&heap));
58     }
59
60     return 0;
61 }
```

Lab8: Heap

- Submit on GitLab
- Write max heap
- Create Lab8 directory on your own GitLab project
- Submit file : source_code(c only, run on linux)
- Filename : StudentID_lab8.c
- Input file : no

Lab8: Heap

- Note
  - Modify part of the Min Heap code to change to the Max Heap

Lab8: Heap

```
donghyeonkim@donghyeonkim-System-Product-Name:~/Hanyang/classes_2020/DS/week8/MaxHeap$ ./a.out
1 : T
2 : S
3 : S
4 : H
5 : P
6 : I
7 : H
8 : E
9 : A
10 : I

T

1 : S
2 : P
3 : S
4 : H
5 : I
6 : I
7 : H
8 : E
9 : A

S
S
P
I
I
H
H
E
A
```