Introduction to Algorithms and Data Structures

6. Data Structure (1) Stack, Queue, and Heap

Professor Ryuhei Uehara,
School of Information Science, JAIST, Japan.
uehara@jaist.ac.jp
http://www.jaist.ac.jp/~uehara

http://www.jaist.ac.jp/~uehara/course/2020/myanmar/

Short Summary So Far

- We have combinations of 3 issues; data structure, what to do, and algorithm
- What to do: E.g., access to the i-th item, search, add/remove/insert
- Array: access in O(1), search in O(n), add tail in O(1), insert/remove in O(n)
- Array in order: search in O(log n), etc.
- Linked list: access in O(n), add/remove in O(1), etc.

- Today's topic: Three abstract data structures for addition and take out

Representative data structures

- Stack: The last added item will be took the first (LIFO: Last in, first out)
- Queue: The first added item will be took the first (FIFO: first in, first out)
- Heap: The smallest item will be took from the stored data

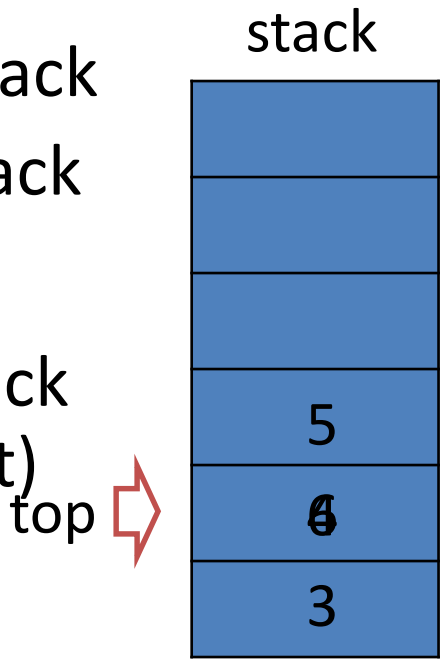
- Implementation of concrete data structures like array and linked list.

- What “Stack” is?
- Implementation by array
- Implementation by linked list

STACK

Stack

- The structure that the last data will be popped first (LIFO: Last in, first out)
- Image: “stack” of dishes, …
- Operations
  - push: add new data into stack
  - pop: take the data from stack
- Pointer
  - top: top element in the stack (where the next item is put)



push 3;
push 4;
push 5;
pop; → 5
pop; → 4
push 6;
pop; → 6

Implementation of stack by an array

- Store a data: push(x)

```
stack[top]=x;
top=top+1;
```

Store the place for the "next" one
Increment the "next" place

- Take the data: pop()

```
top=top-1;
return stack[top];
```

Decrement the "next" place
Return the current one

- What kind of errors?
  - Overflow: push (x) when top == size(stack)
  - Underflow: pop() when top == 0

Implementation of stack by an array

```
class Stack {
    private int[] data;
    private int top;
    public Stack (int maxsize) {
        data = new int[maxsize];
        top=0;
    }
    public void push(int x) {
        if (top < data.Length) {
            data[top] = x;
            top ++;
        } else {
            System.Console.WriteLine("overflow");
        }
    }
    public int pop() {
        if (top > 0) {
            top --;
            return data[top];
        } else {
            System.Console.WriteLine("underflow");
            return -1;
        }
    }
    public void print() {
        for (int i=0; i<top; i++) {
            System.Console.Write(data[i]+" ");
        }
        System.Console.WriteLine(" top <- "+top);
    }
}
```

Initialization

Exercise: Resize the array when it overflows

Display for check

```
public class i111_06_p7 {
    public static void Main () {
        Stack st = new Stack(6);
        st.push(3);
        st.print();
        st.push(4);
        st.print();
        st.push(5);
        st.print();
        System.Console.WriteLine("pop="+st.pop());
        st.print();
    }
}
```

Make it of size 6

Output the result of pop

Implementation of stack by a linked list

```
class StackLL {
    private Node top;

    public StackLL () {
        top=null;
    }

    public void push(int x) {
        Node n = new Node(x, top);
        top = n;
    }

    public int pop() {
        if (top != null) {
            int topvalue = top.data;
            top = top.next;
            return topvalue;
        } else {
            System.Console.WriteLine("underflow");
            return -1;
        }
    }

    public void print() {
        Node n = top;
        while (n != null) {
            System.Console.Write(n.data+" ");
            n = n.next;
        }
        System.Console.WriteLine(" top -> ");
    }
}
```

Top node

No need to check overflow

```
public class i111_06_p8 {
    public static void Main () {
        StackLL st = new StackLL();
        st.push(3);
        st.print();
        st.push(4);
        st.print();
        st.push(5);
        st.print();
        System.Console.WriteLine("pop="+st.pop());
        st.print();
    }
}
```

Size is not needed

```
class Node {
    public int data;
    public Node next;
    public Node(int i, Node n) {
        this.data = i;
        this.next = n;
    }
}
```

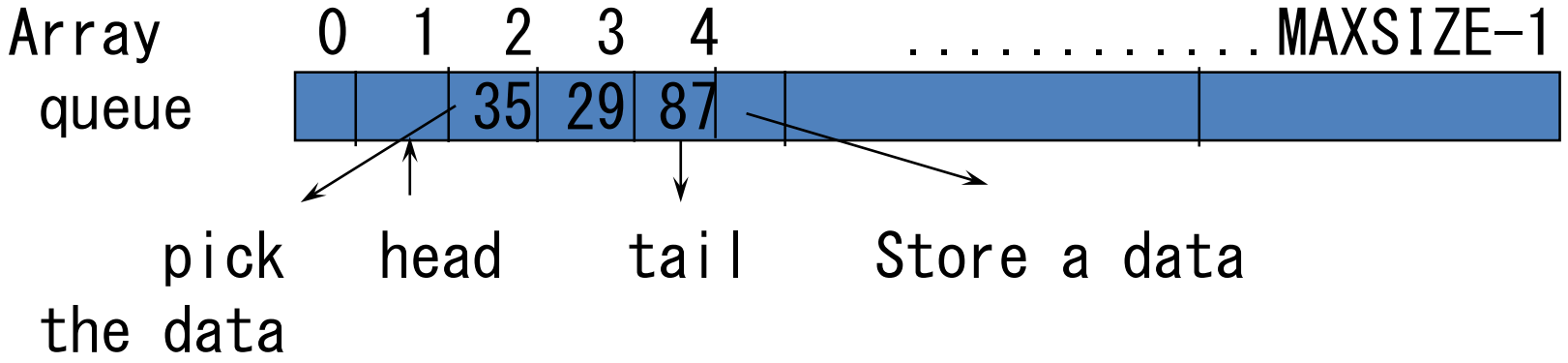
“Queue” means “(waiting) line”

QUEUE

Queue

- The first data will be took first

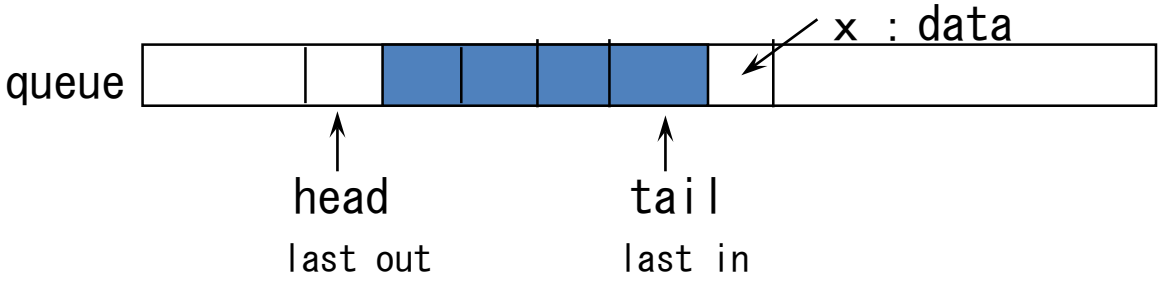
  (FIFO: first in, first out)
- Image: In front of famous restaurant



Data are stored in from queue[head+1] to queue[tail]

You may feel it is not intuitive since it is not from queue[head] to queue[tail]…

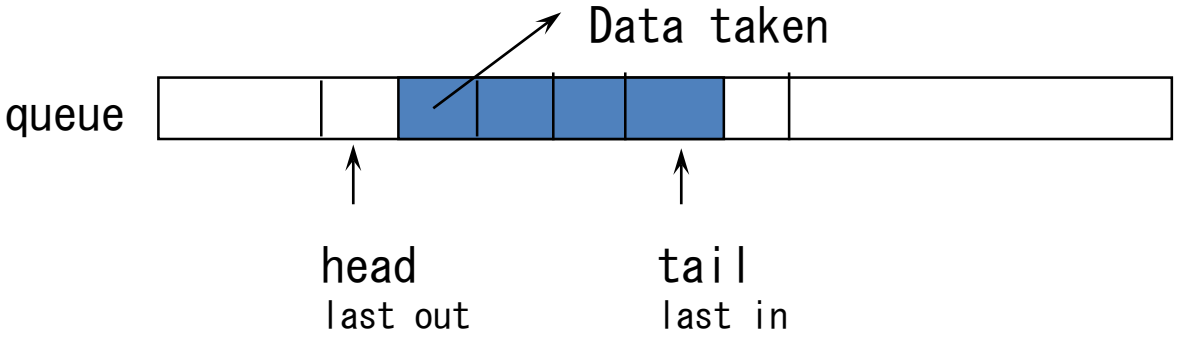
Simple implementation of queue by an array: Add a data




```
void append(int x){
  tail = tail + 1;
  queue[tail] = x;
}
```

Move to the place to put
(and place put the last)

Simple implementation of queue by an array: Take a data




```
int get(){
    head = head + 1;
    return queue[head];
}
```

Move to the place to be taken
(and place taken the last)

Problem of simple implementation of queue: Waste area…

- What happens when we use queue as follows?

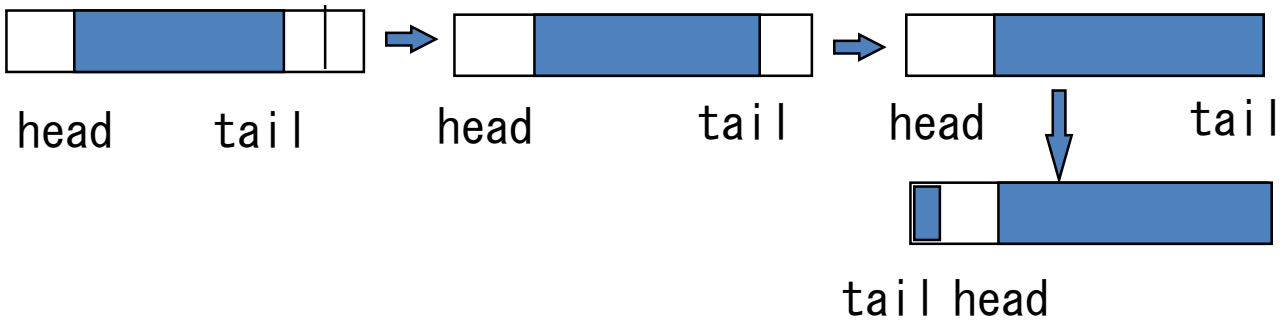
```
int get(){
  head = head + 1;
  return queue[head];
}
```

```
int queue[MAX_SIZE];
int head, tail;
void main(){
  head=0; tail=0;
  append(3); get();
  append(4); get();
}
```

```
void append(int x){
  tail = tail + 1;
  queue[tail] = x;
}
```

get() append(4)

Solution: Use array *cyclic*




```
void append(int x){
  tail = (tail + 1) % MAXSIZE;
  queue[tail] = x;
}
int get(){
  head = (head + 1) % MAXSIZE;
  return queue[head];
}
```

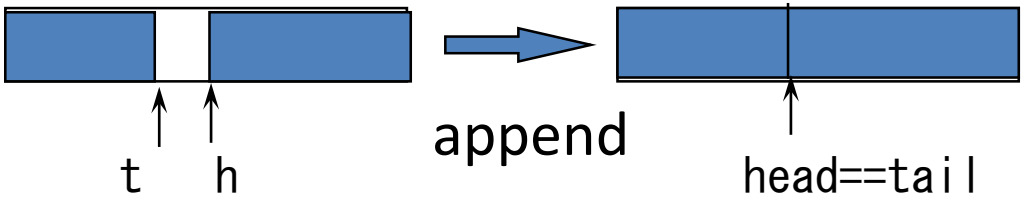
Return to 0

Return to 0

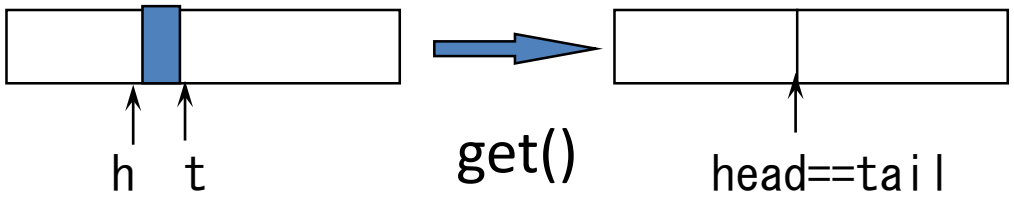
Problem of queue in cyclic array:

We cannot distinguish between full and empty

When it is full;



When it is empty;



In both cases, we have head==tail.

We may count the number, but it is not a good situation to be full…

Solution: We define full when we have tail==head when append.

```
void append(int x){
  tail = (tail + 1) % MAXSIZE;
  queue[tail] = x;
  if(tail == head) printf("Queue Overflow ");
}
int get(int x){
  if(tail == head) printf("Queue is empty ");
  else {
    head = (head + 1) % MAXSIZE;
    return queue[head];
  }
}
```

Implementation of Queue in C#

```
class Queue {
    private int[] queue;
    private int head;
    private int tail;
    public Queue (int maxsize) {
        queue = new int[maxsize];
        head = 0;
        tail = 0;
    }
    public void append(int x) {
        tail = (tail +1) % queue.Length;
        queue[tail] = x;
        if (tail == head) {
            System.Console.WriteLine("overflow");
        }
    }
    public int get() {
        if (tail == head) {
            System.Console.WriteLine("underflow");
            return -1;
        }
        head = (head + 1) % queue.Length;
        //int t=queue[head]; queue[head]=0; return t;
        return queue[head];
    }
    public void print() {
        for (int i=0; i<queue.Length; i++) {
            System.Console.Write(queue[i]+" ");
        }
        System.Console.WriteLine("h"+head+" t"+tail);
    }
}
```

It is better to initialize by -1

full!!

empty!!

※Erase the taken data

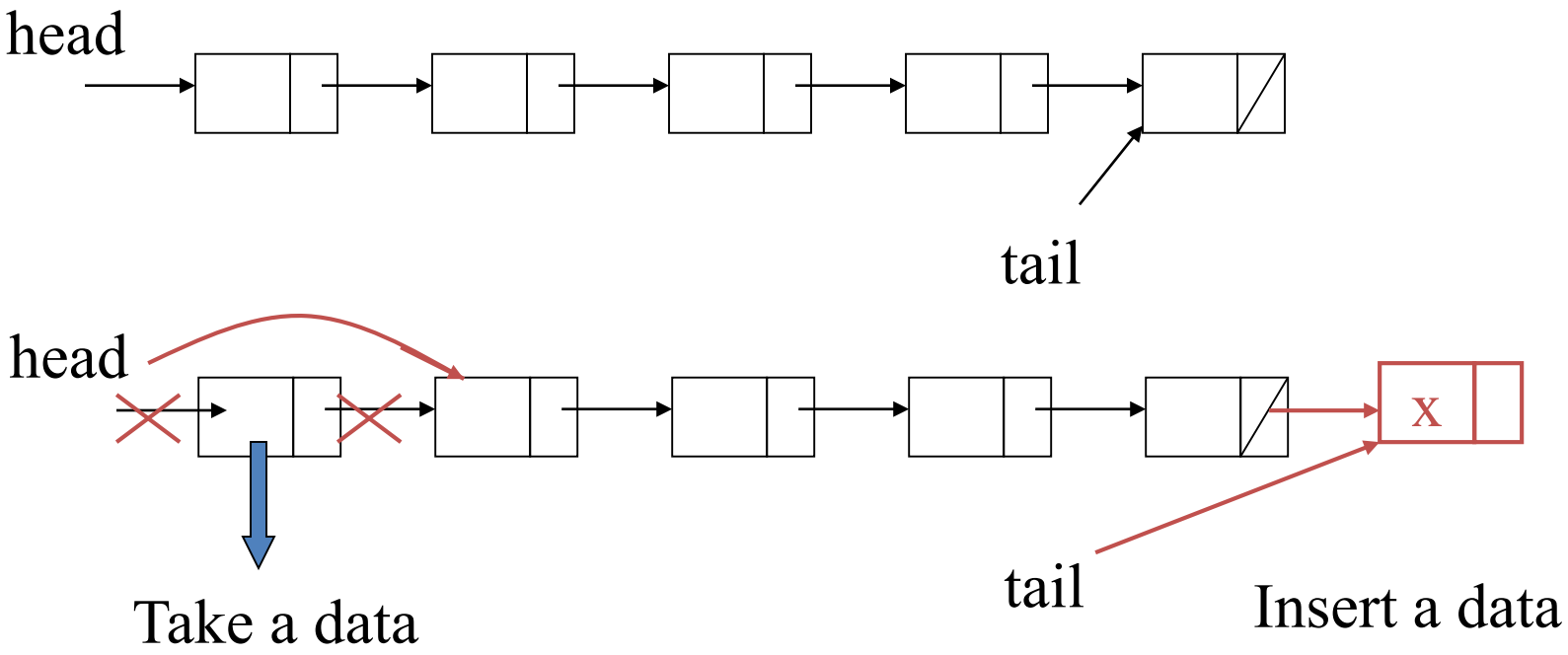
```
public class i111_06_p16 {
    public static void Main () {
        Queue qu = new Queue(3);
        qu.append(3); qu.print();
        qu.append(4); qu.print();
        System.Console.WriteLine("get="+qu.get());
        qu.print();
        qu.append(5); qu.print();
        System.Console.WriteLine("get="+qu.get());
        qu.print();
        qu.append(6); qu.print();
    }
}
```

Make it size 3

Output result by get

Implementation of queue by linked list

Insertion of a data : From tail of the list: pointer tail
Take a data : From top of the list: pointer head



Exercise: Make program by yourself!

Implementation of Queue in C#

```
class QueueLL {
    private Node head;
    private Node tail;
    public QueueLL () {
        head = null;
        tail = null;
    }
    public void append(int x) {
        Node n = new Node(x,null);
        if (head == null) {
            head = n;
        } else {
            tail.next = n;
        }
        tail = n;
    }
    public int get() {
        if (head == null) {
            System.Console.WriteLine("underflow");
            return -1;
        }
        int headValue = head.data;
        head = head.next;
        return headValue;
    }
    public void print() {
        Node n = head;
        while (n != null) {
            System.Console.Write(n.data+" ");
            n = n.next;
        }
        System.Console.WriteLine(" head -> tail");
    }
}
```

Ex: What happens?

Try to take data from empty

Size is not needed

```
public class i111_06_p18 {
    public static void Main () {
        QueueLL qu = new QueueLL();
        qu.append(3); qu.print();
        qu.append(4); qu.print();
        System.Console.WriteLine("get="+qu.get());
        qu.print();
        qu.append(5); qu.print();
        System.Console.WriteLine("get="+qu.get());
        qu.print();
        qu.append(6); qu.print();
    }
}
```

```
class Node {
    public int data;
    public Node next;
    public Node(int i, Node n) {
        this.data = i;
        this.next = n;
    }
}
```

- “heap” also means “stack”, but more “mountain”-like shape?
- Simple implementation by array
- Implementation by array using an idea of binary tree

HEAP

Heap

- Add/remove data
- Elements can be taken from minimum (or maximum) in order

Q. How can we implement?

Implement of heap (1): Simple implemer

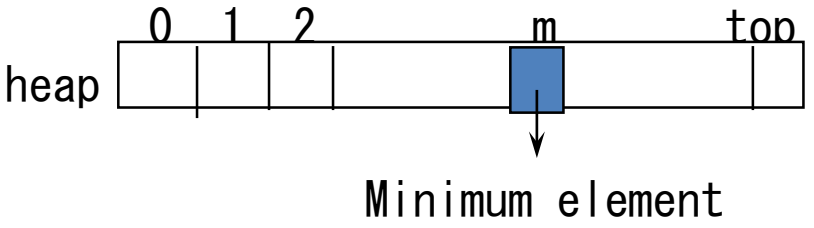
An array heap[] and top, the number of data

- Initialize: top = 0
- Insert data:

```
heap[top] = x;
top = top + 1;
```

- Take minimum one: Find the minimum element heap[m] in heap[] and output. Then copy heap[top-1] to heap[m], and decrease top by 1.

```
m = 0;
for(i=1; i<top; i++)
  if(heap[i] < heap[m])
    m = i;
x = heap[m];
heap[m] = heap[top-1];
top = top - 1;
return x;
```



Problem of simple implementation: Slow for reading data

- Store: O(1) time

```
heap[top++]=x
```

- Take: O(n) time

```
m = 0;
for(i=1; i<top; i++)
  if(heap[i] < heap[m])
    m = i;
x = heap[m];
heap[m] = heap[top-1];
top = top - 1;
return x;
```

← As same as

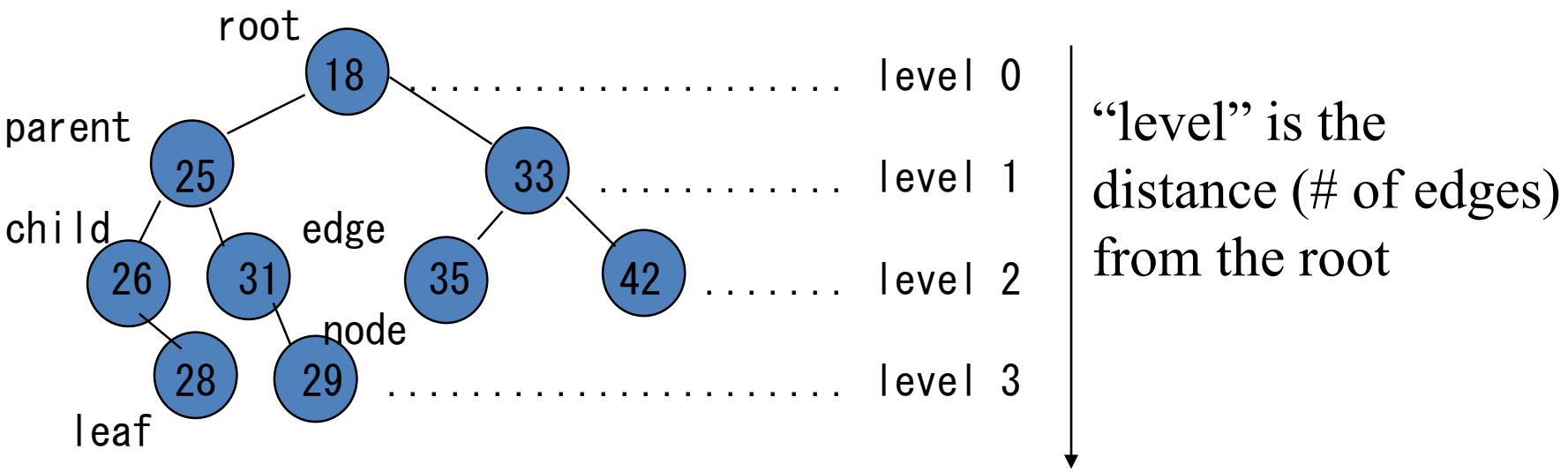
```
heap[top] = x;
top = top + 1;
```

in C

【Important】

【Note: different from binary search tree!】

Implementation of heap by binary tree



root: node that has no parent

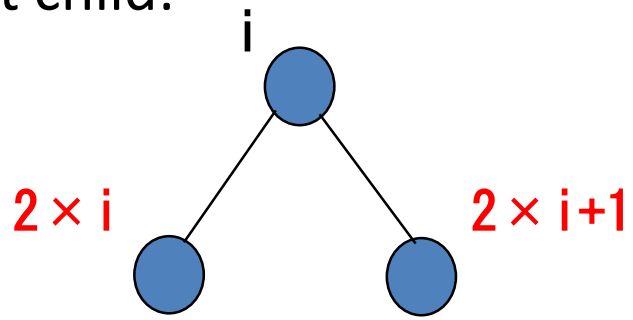
leaf: node that has no child

A tree is called a *binary tree*

if each node has at most 2 children

Property of binary tree for heap

1. Assign 1 to the root.
2. For a node of number i, assign 2 × i to the left child and assign 2 × i+1 to the right child:



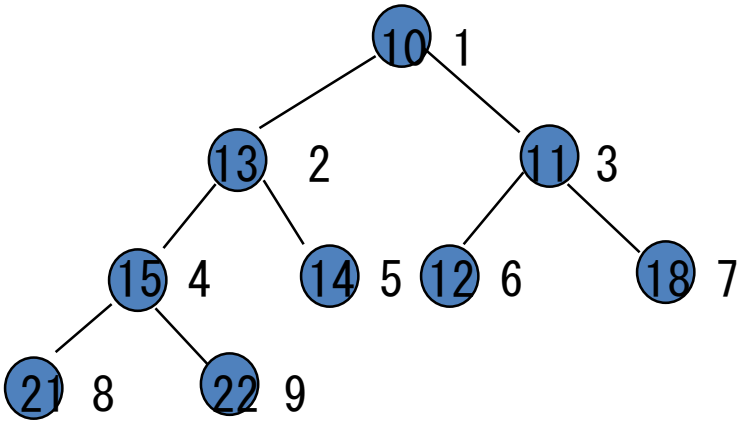
3. No nodes assigned by the number greater than n.
4. For each edge, parent stores data smaller than one in child.

The maximum level of heap: $[\log_2 (n+1) - 1]$

Each node has a unique path from the root, and its length is O(log n).

※Some textbooks prefer to start from 0 instead of 1. In this case, considering children as 2i+1 and 2i+2, we have the same structure.

Example of a heap by binary tree



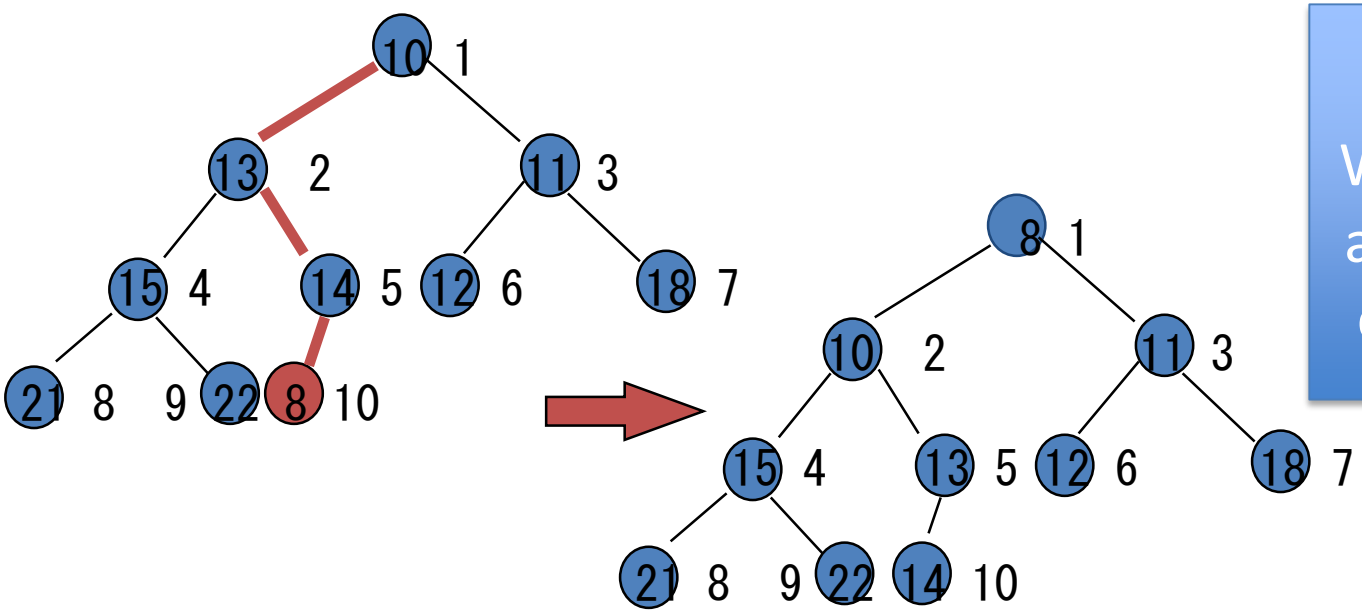
1. Assign 1 to the root.
2. For a node of number i, assign $2 \times i$ to the left child and assign $2 \times i+1$ to the right child.
3. No nodes assigned by the number greater than n.
4. For each edge, parent stores data smaller than one in child.

We can use an array, instead of linked list!

1	2	3	4	5	6	7	8	9
10	13	11	15	14	12	18	21	22

Add a data to heap

(1) temporally, add data to node n+1 (n+1st element in array)
(2) traverse to the root step by step, and
    if parent > child then swap parent and child

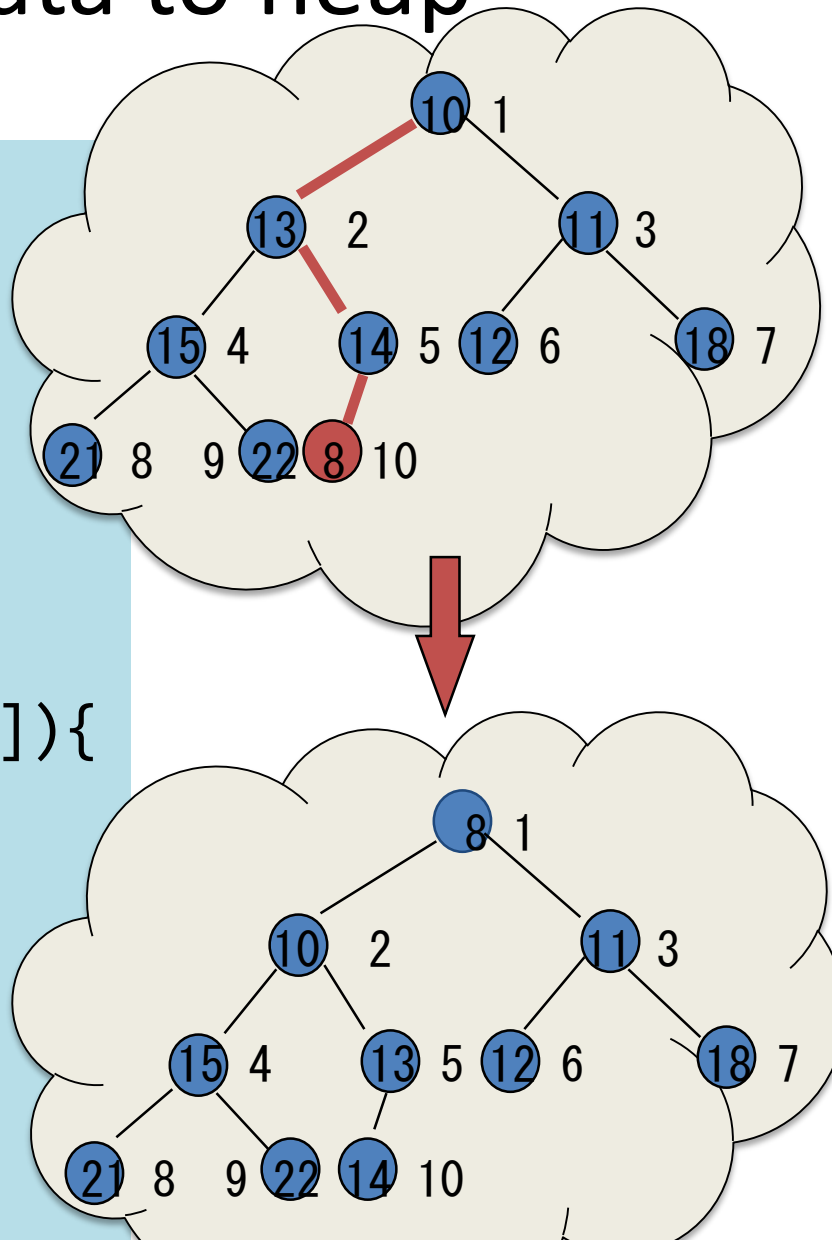


Exercise:
Why does this algorithm has consistency?

That is, from n+1st node to the root, the data are in order. This algorithm does not occur any problem with any other part of tree.

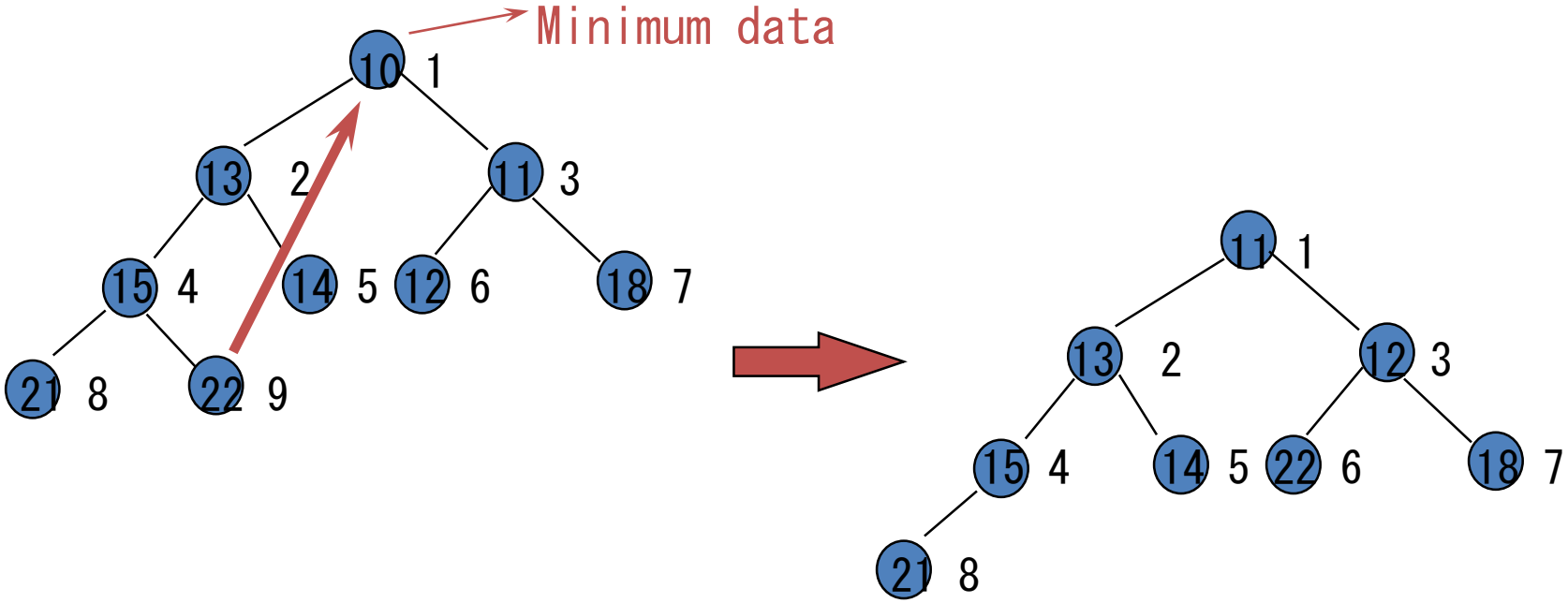
Program for adding a data to heap

```
void pushHeap(int x){
  int i, j;
  if(++n >= MAXSIZE)
    stop("Heap Overflow");
  else{
    heap[n] = x;
    i=n; j=i/2;
    while(j>0 && x < heap[j]){
      heap[i] = heap[j];
      i=j; j=i/2;
    }
    heap[i] = x;
  }
}
```



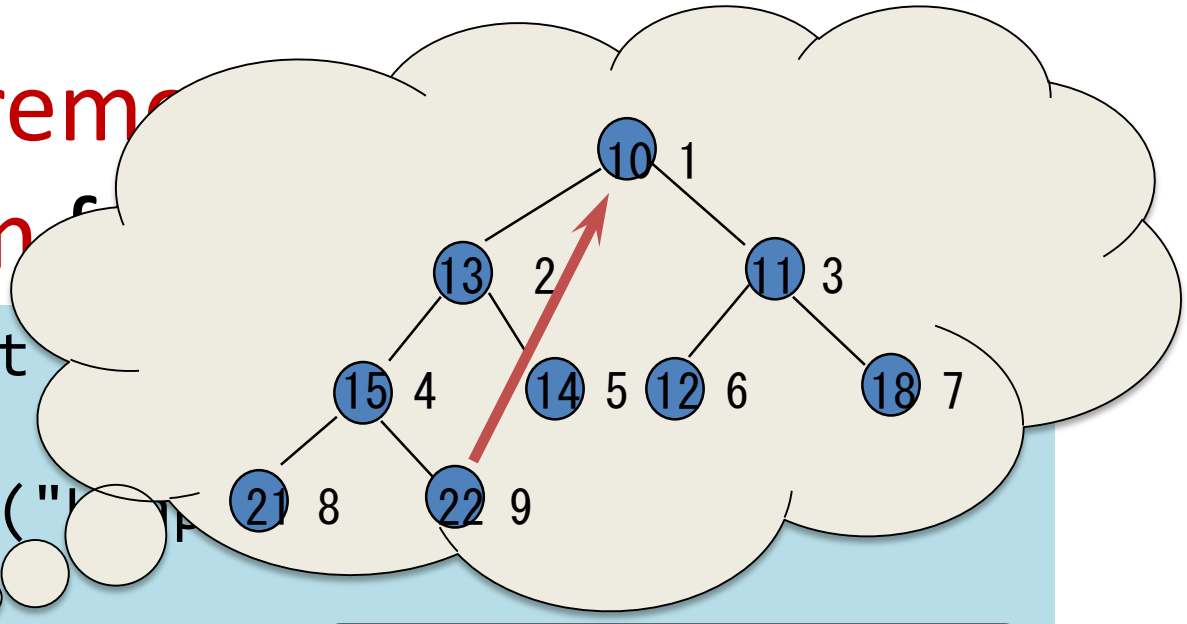
Heap: Take the minimum item

(1) Take the minimum data at the root
(2) Copy the last item (of number n) to the root
(3) Traverse from the root to a leaf as follows
For each pair of two children, choose the smaller one,
and exchange parent and child if child is smaller than parent.

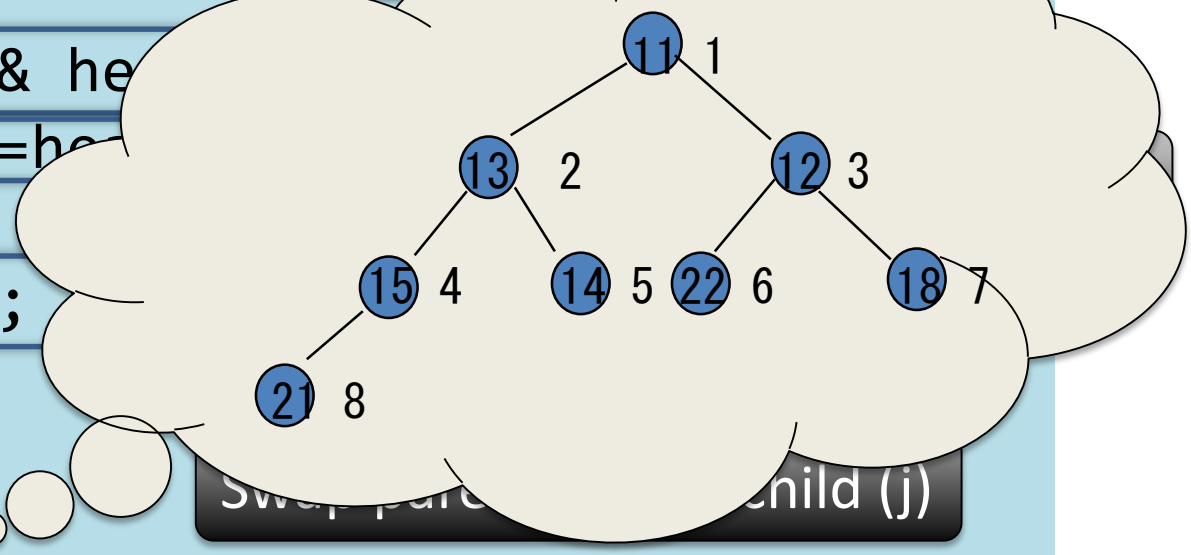


Program for remo
item f

```
int* deleteMin(int
  int x, i, j, t;
  if(n == 0) stop("
  else{
    heap[1]=heap[n--];
    for(i=1;i*2<=n;i=j){
      j=i*2;
      if(j+1<=n && he
      if(heap[i]<=he
      else {
        t=heap[i];
      }
    }}
  return heap;}
```



Node i has child &&
right child may be smaller



Swap parent and child (j)

Time complexity of binary heap

- Let n be the size of heap
  - Addition: O(log n)
  - Removal: O(log n)

  - Each operation runs in time proportional to the depth of the heap
  - The depth of heap is almost log n

Summary

- Stack: Structure that the last data will be popped first (LIFO: Last in, first out)
- Queue: The first data will be took first

  (FIFO: first in, first out)
- Heap: Elements are taken from minimum in order

- Implemented by array/linked list
- Heap is efficient by using binary tree

Q: How about heap by linked list?