# Introduction to Algorithms and Data Structures

## Lesson 9: Data structure (3) Stack, Queue, and Heap

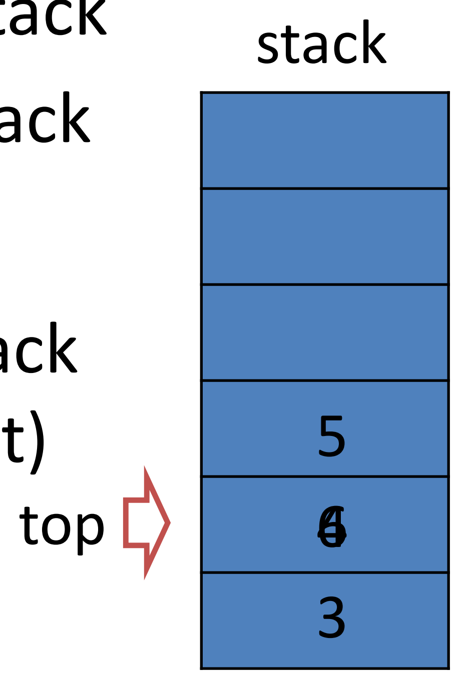
Professor Ryuhei Uehara,
School of Information Science, JAIST, Japan.
uehara@jaist.ac.jp
http://www.jaist.ac.jp/~uehara

# Representative data structure

- 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

# Stack

- The structure that the last data will be popped first (LIFO: Last in, first out)
- 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;
```

- Take the data: pop()

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

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

# Implementation of stack by an array

```
int stack[MAXSIZE];
int top = 0;
void push(int x){
   if(top < MAXSIZE){
      stack[top] = x; top = top + 1;
   } else
      printf("STACK overflow");
}
int pop(){
   if(top > 0){
      top = top - 1; return stack[top];
   } else
      printf("STACK underflow");
}
```

# Implementation of stack by linked list

- Point: You don't need to fix the size of stack

```
typedef struct{
  int data; struct list_t *next;
}list_t;
```

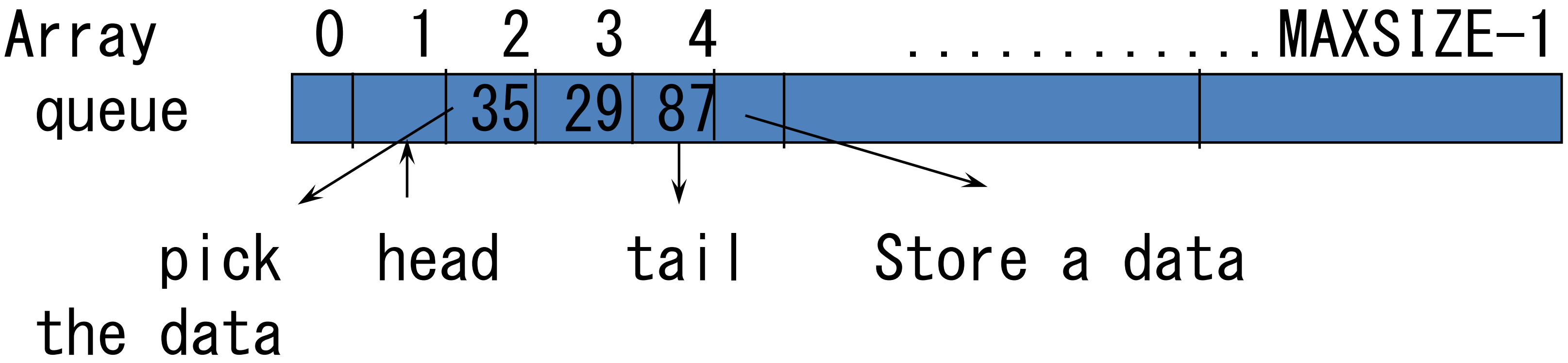
```
list_t* push(list_t *top,int x){
  list_t *ptr;
  ptr=(struct list_t*) malloc(sizeof(list_t));
  ptr->data=x; ptr->next=top; return ptr;
}
list_t* pop(list_t *top){
  list_t *ptr; ptr=top->next; free(top); return ptr;
}
```

It is not necessary if the language has garbage collection

# Queue

- The first data will be took first

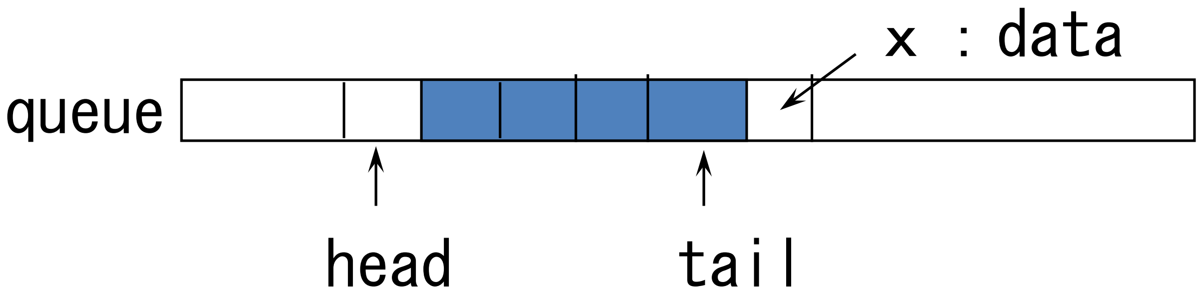
  (FIFO: first in, first out)




Data are stored in

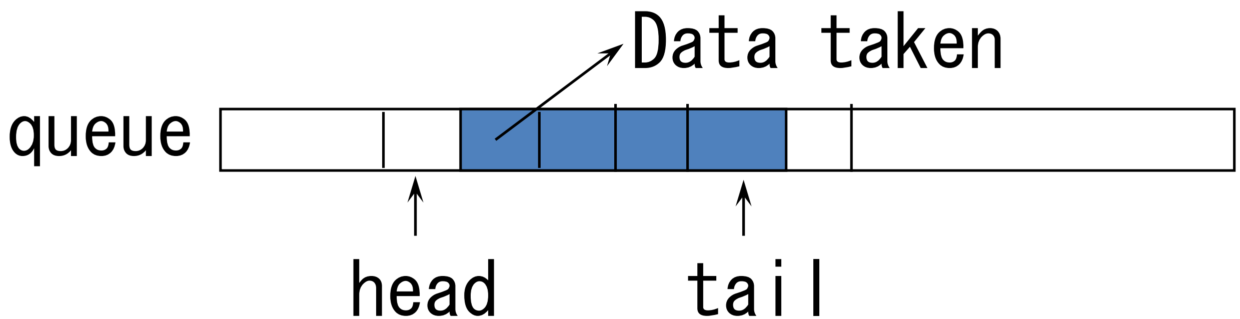
from queue[head+1] to queue[tail]

# Add a data into queue




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

# Simple implementation of queue by array: take a data




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

# Problem of simple implementation of queue: Waste area…

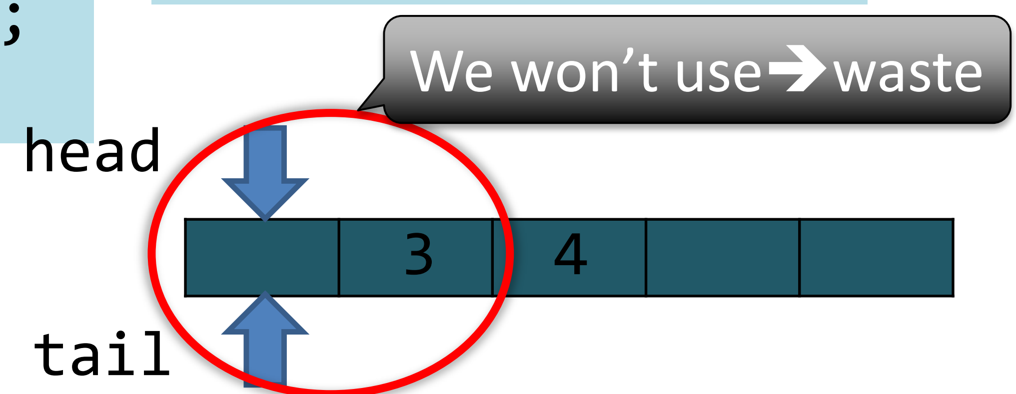
- What happens when we use queue as follows?

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

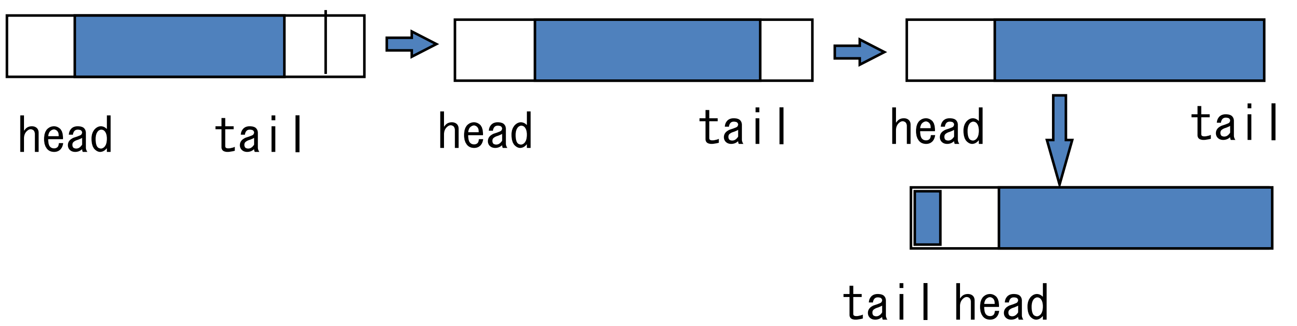
```
int queue[MAX_SIZE];
int head, tail;
void main(){
  head=0; tail=0;
  append(3); get();
  append(4); get();
}
```

```
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];
}
```

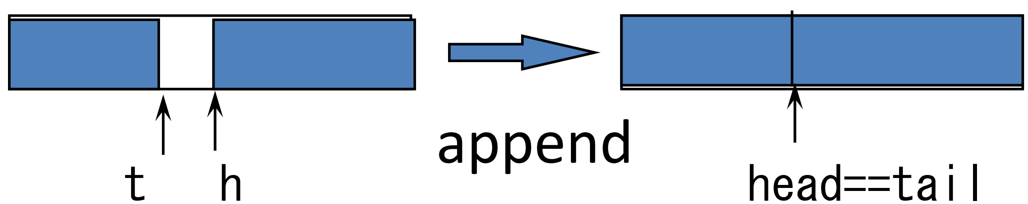
Return to 0

Return to 0

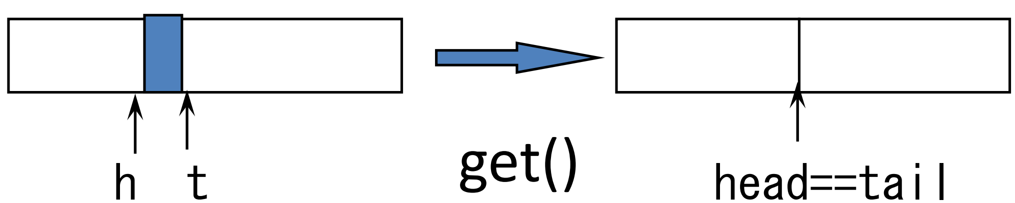
# Problem of queue in cyclic array:

## We cannot distinguish between (full) and (empty)

When it is full;




When it is empty;




In both cases, we have head==tail.

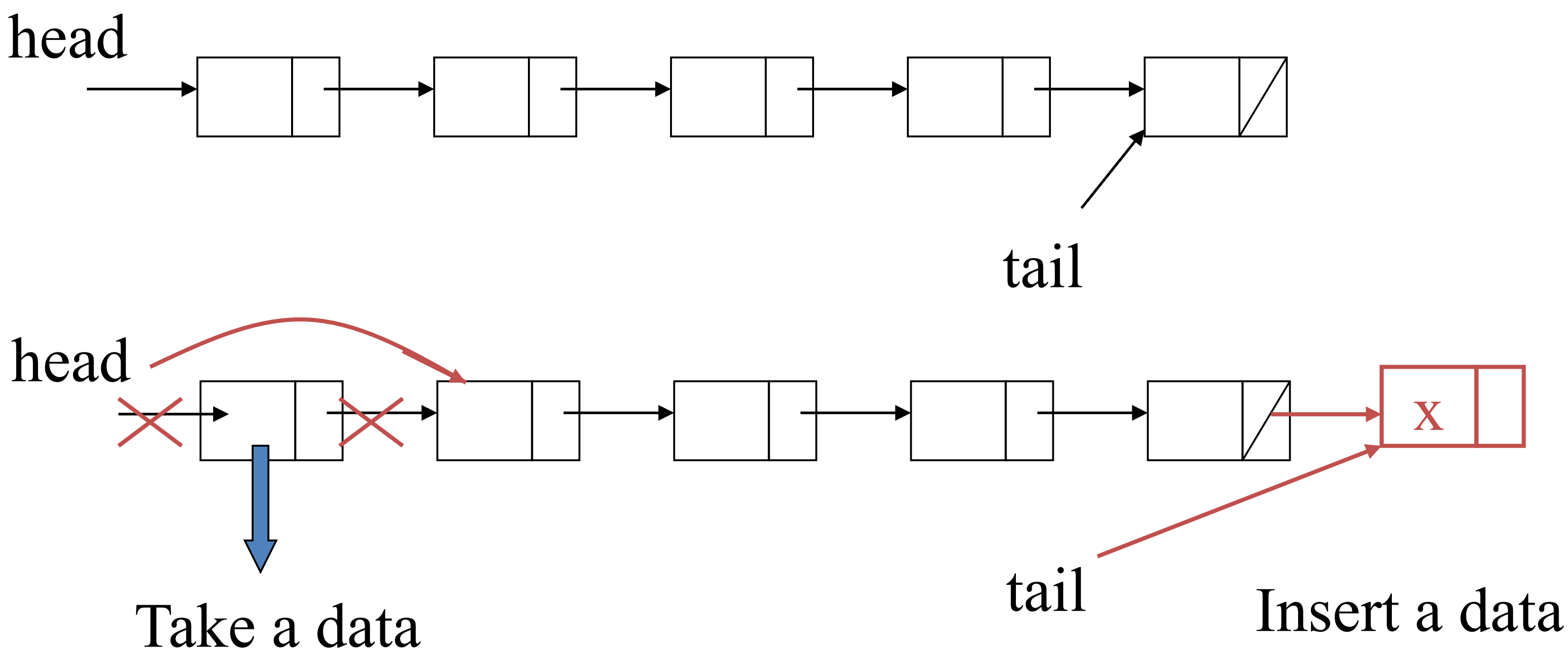
# Solution: We define (full) when we have tail==head when append.

```
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 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!

# Heap

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

**Q. How can we implement?**

# Implement of heap (1): Simple implemen

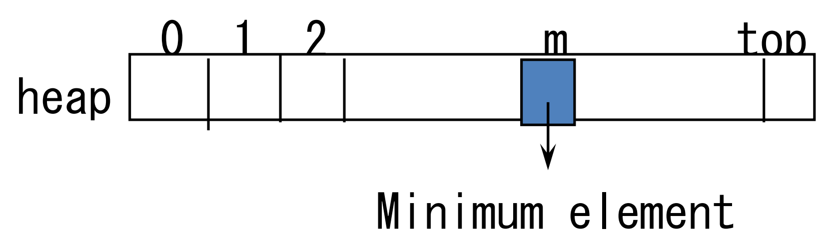
An array heap[] and top, the number of data

- 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

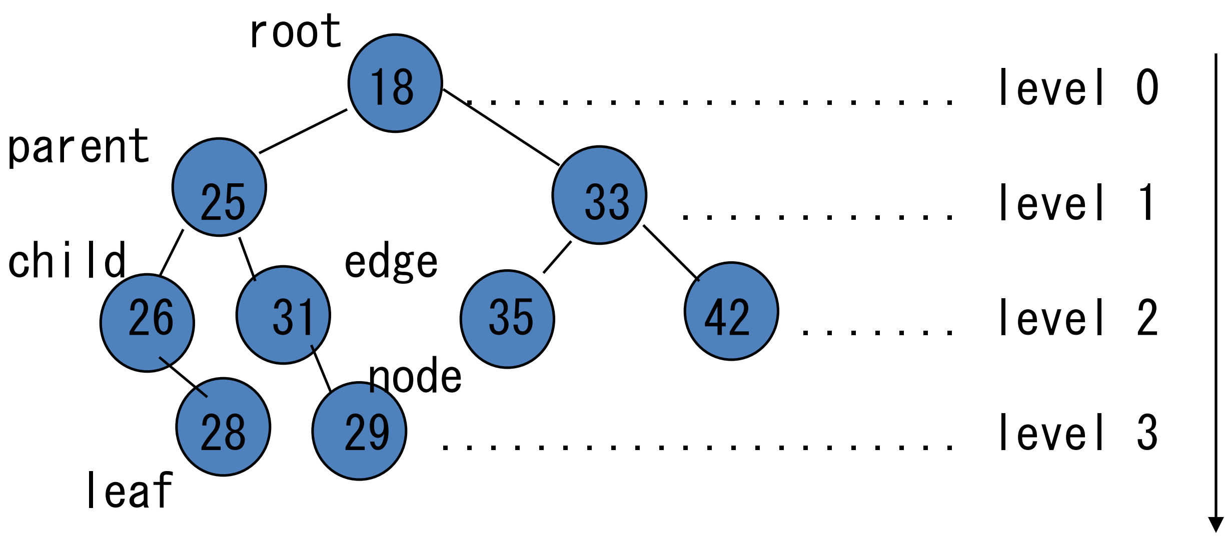
- Store: O(1)

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

- Take: O(n)

```
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;
```

# Heap by binary tree




"level" is the distance (# of edges) from the root

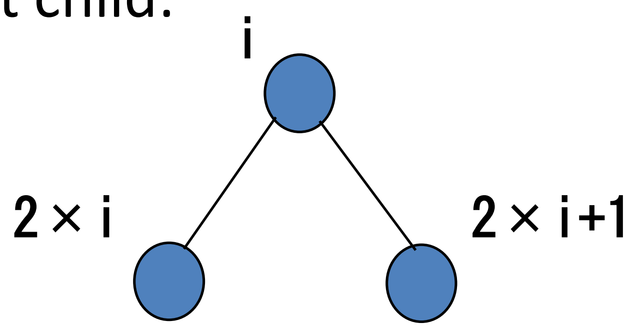
**root**：node that has no parent

**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:

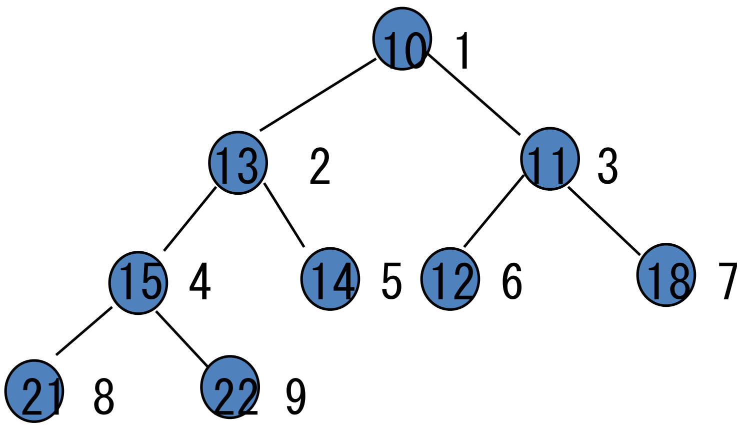



3. No nodes assigned by the number greater than n.
4. For each edge, parent stores data smaller than one in child.

**The maximum level of heap: ceil( $\log_2(n+1) - 1$ )**

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

# Example of a heap by binary tree



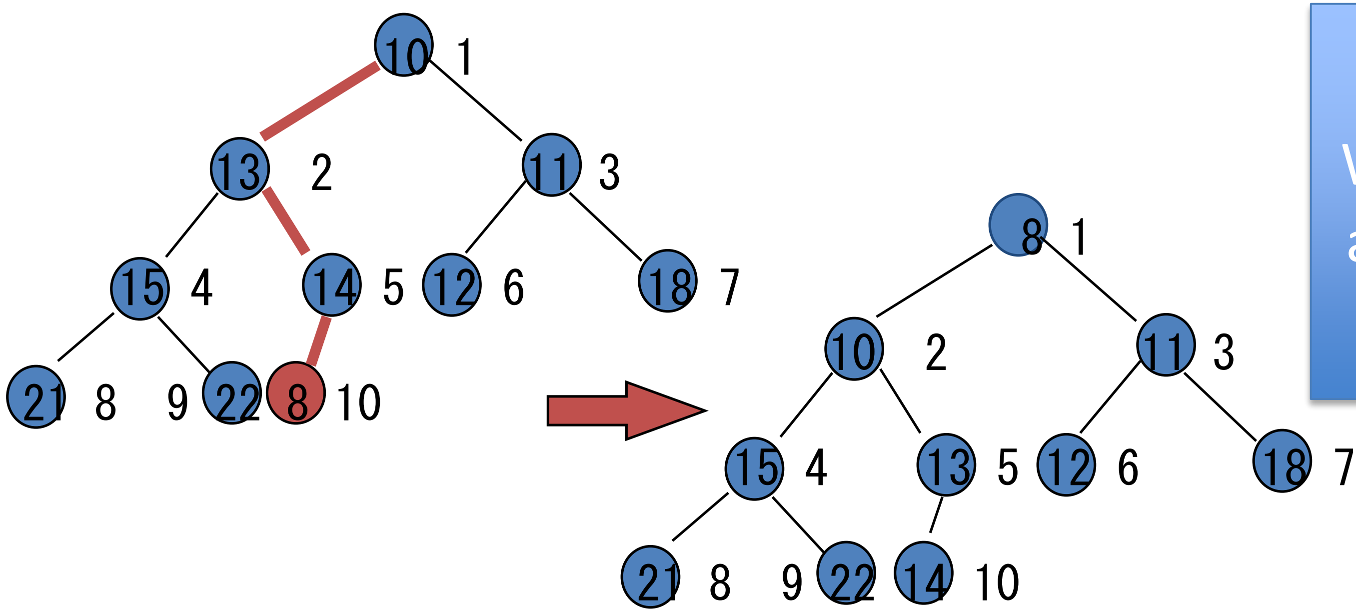

1. Assign 1 to the root.
2. For a node of number i, assign 2×i to the left child and assign 2×i+1 to the right child.
3. No nodes assigned by the number greater than n.
4. For each edge, parent stores data smaller than one in child.

## 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

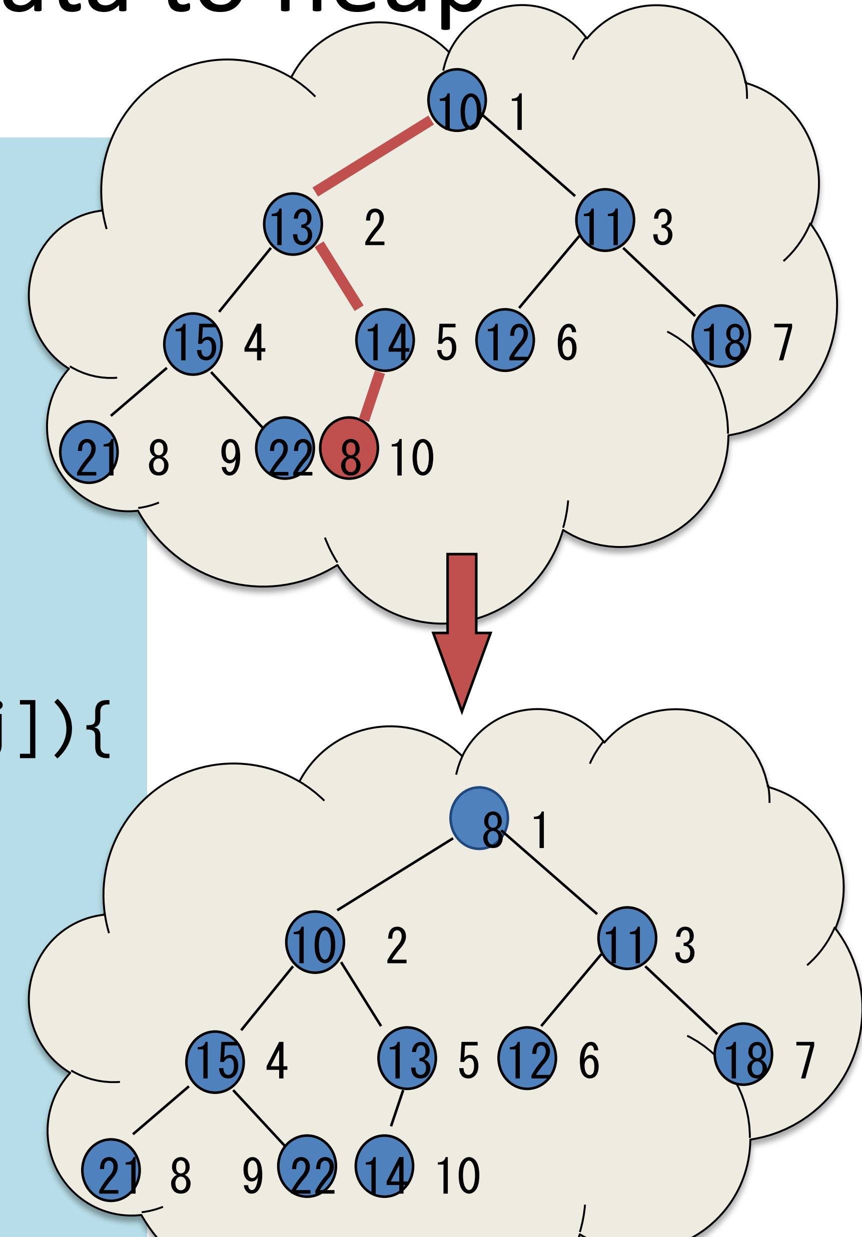



Report 3:
Why does this algorithm has consistency?

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.

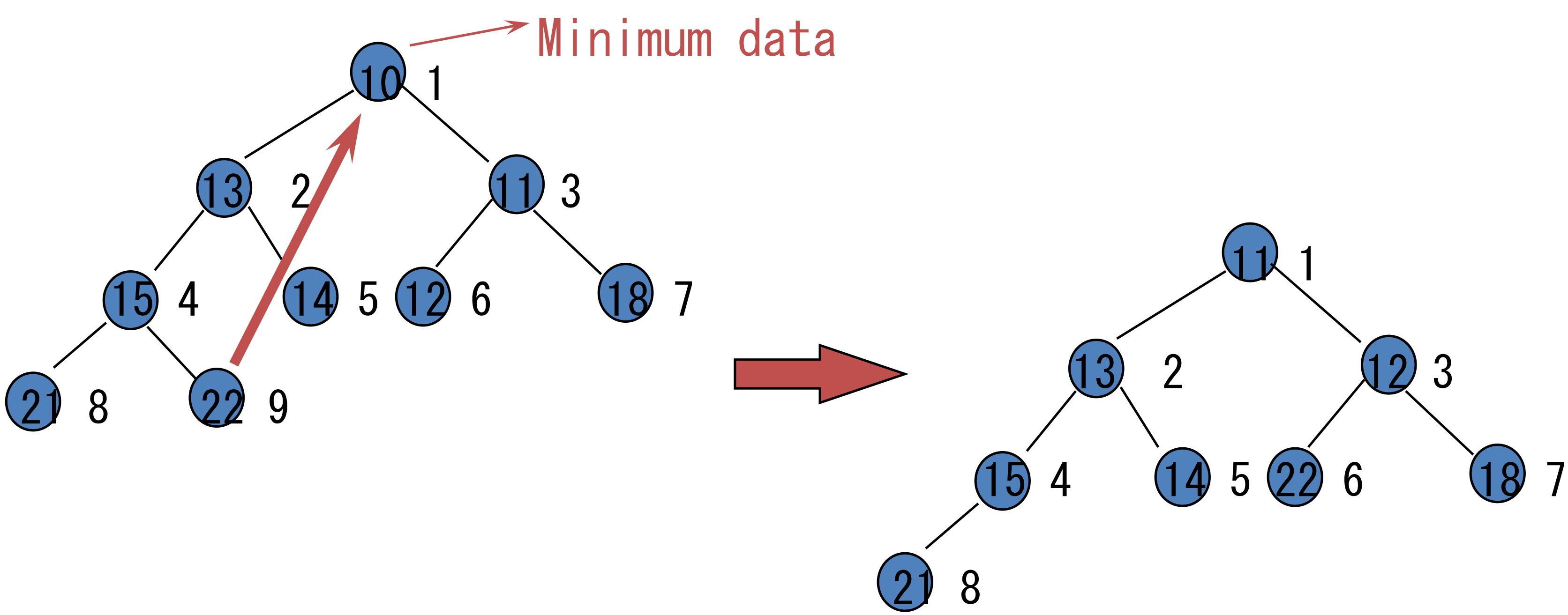
# Program for adding a data to heap

```
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;
  }
}
```



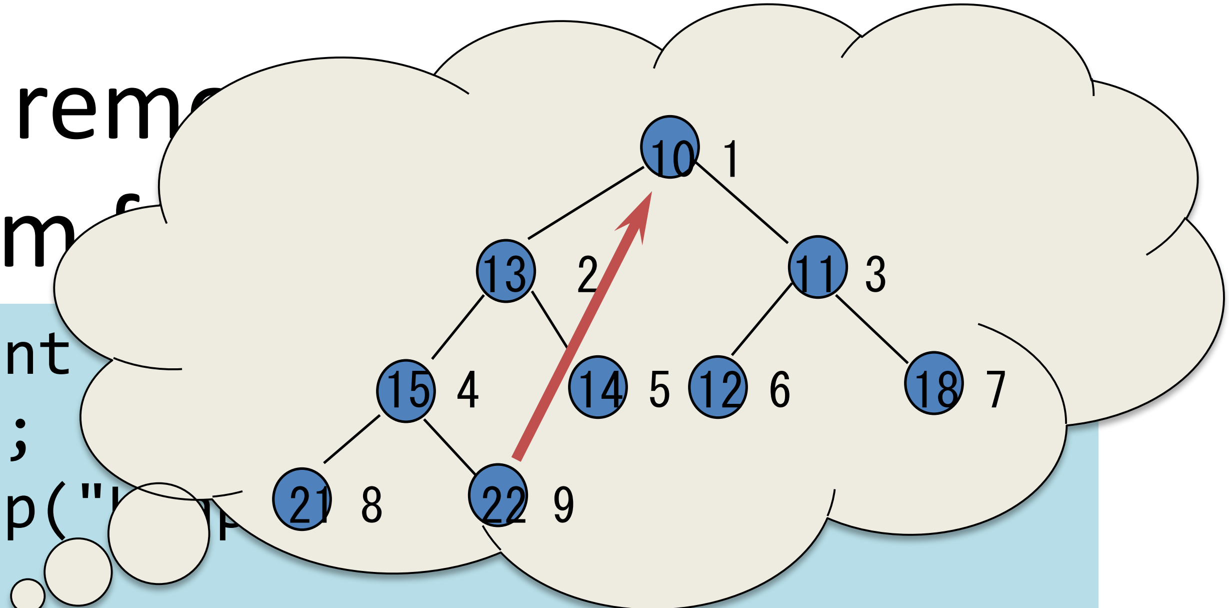
# Heap: Take the minimum item

(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.

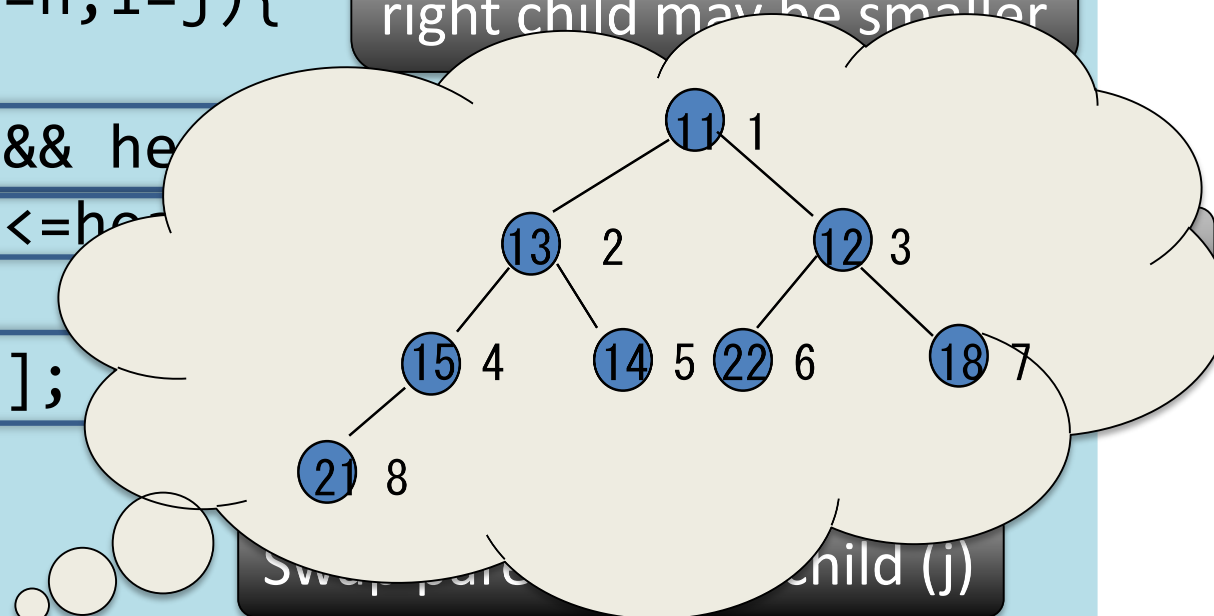


# Program for rem… item f…

```
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




…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 O(log n)

# Report Problem 3

1. Explain what does a “pointer” indicate in RAM mod … binary sear…

Choose and answer one of the following two problems

2. In the procedure of a binary search tree, show that those two procedures keep the assertion of binary search tree.

3. Explain why do addition/removal of data to/from binary heap work. Especially, show the procedures do not break consistency.