CS711008Z Algorithm Design and Analysis

Lecture 7. Binary heap, binomial heap, and Fibonacci heap

Dongbo Bu

Institute of Computing Technology
Chinese Academy of Sciences, Beijing, China

Outline

- Introduction to priority queue
- Various implementations of priority queue:
  - Linked list: a list having n items is too long to support efficient EXTRACTMIN and INSERT operations simultaneously;
  - Binary heap: using a **tree** rather than a **linked list**;
  - Binomial heap: allowing **multiple trees** rather than **a single tree** to support efficient UNION operation
  - Fibonacci heap: implement DECREASEKEY via simply **cutting an edge** rather than **exchanging nodes**, and control a "bushy" tree shape via allowing **at most one child losing** for any node.

Priority queue

Priority queue: motivation

- Motivation: It is usually a case to **extract the minimum** from a set S of n numbers, **dynamically**.
- Here, the word "dynamically" means that on S, we might perform INSERTION, DELETION and DECREASEKEY operations.
- The question is how to organize the data to efficiently support these operations.

Priority queue

- **Priority queue** is an **abstract data type** similar to stack or queue, but each **element** has a **priority** associated with its **name**.
- A min-oriented priority queue must support the following core operations:
  1. H=$\textsc{MakeHeap}()$: to create a new heap H;
  2. $\textsc{Insert}(H, x)$: to insert into H an element x together with its priority
  3. $\textsc{ExtractMin}(H)$: to extract the element with the highest priority;
  4. $\textsc{DecreaseKey}(H, x, k)$: to decrease the priority of element x;
  5. $\textsc{Union}(H_1, H_2)$: return a new heap containing all elements of heaps H_1 and H_2, and destroy the input heaps

Priority queue is very useful

- Priority queue has extensive applications, such as:
  - Dijkstra's shortest path algorithm
  - Prim's MST algorithm
  - Huffman coding
  - A^* searching algorithm
  - HeapSort
  - ......

An example: Dijkstra's algorithm

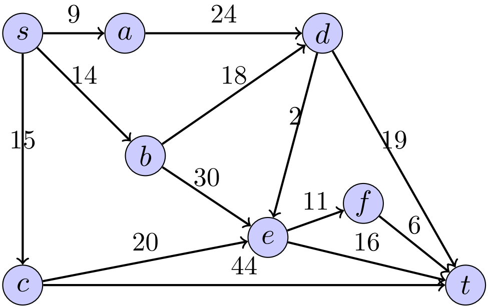
Dijkstra's algorithm [1959]

$\textsc{Dijkstra}(G, s, t)$

$key(s) = 0$; $//key(u)$ stores an upper bound of the shortest distance from s to u;
$PQ.\ \textsc{Insert}\ (s)$;
for all node $v \neq s$ **do**
    $key(v) = +\infty$
    $PQ.\ \textsc{Insert}\ (v)$ $//n$ times
end for
$S = \{\}$; // Let S be the set of explored nodes;
while $S \neq V$ **do**
    $v^* = PQ.\ \textsc{ExtractMin}()$; $//n$ times
    $S = S \cup \{v^*\}$;
    for all $v \notin S$ and $< v^*, v > \in E$ **do**
        if $key(v) + d(v^*, v) < key(v)$ **then**
            $PQ.\textsc{DecreaseKey}(v, key(v^*) + d(v^*, v))$; $//m$ times
        end if
    end for
end while

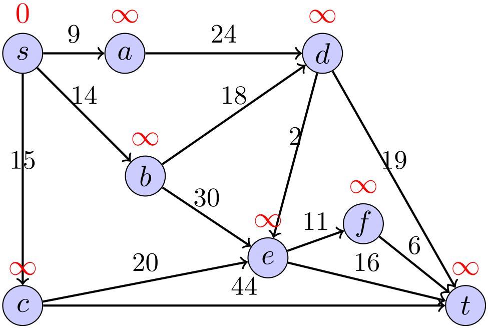
Here PQ denotes a min-priority queue.

Dijkstra's algorithm: an example



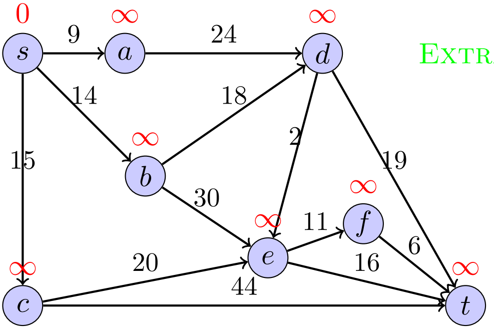
Dijkstra's algorithm: an example

$$S = \{\}$$
$$PQ = \{s(0), a(\infty), b(\infty), c(\infty), d(\infty), e(\infty), f(\infty), t(\infty)\}$$



Dijkstra's algorithm: an example

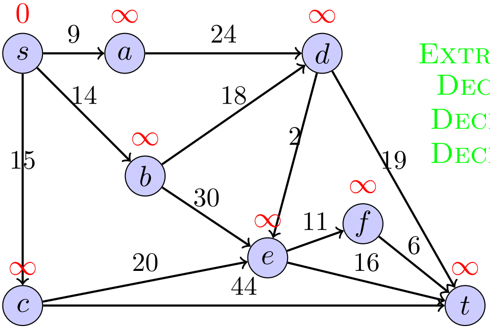
$$S = \{\}$$
$$PQ = \{s(0), a(\infty), b(\infty), c(\infty), d(\infty), e(\infty), f(\infty), t(\infty)\}$$



ExtractMin returns s

Dijkstra's algorithm: an example

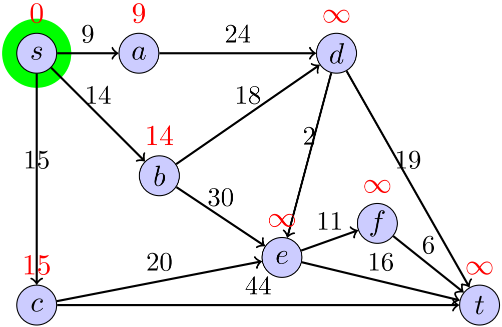
$$S = \{\}$$
$$PQ = \{s(0), a(\infty), b(\infty), c(\infty), d(\infty), e(\infty), f(\infty), t(\infty)\}$$



ExtractMin returns s
DecreaseKey$(a, 9)$
DecreaseKey$(b, 14)$
DecreaseKey$(c, 15)$

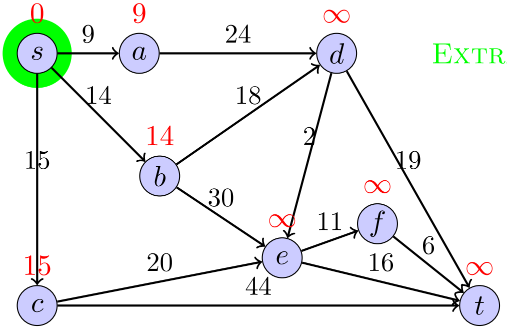
Dijkstra's algorithm: an example

$$S = \{s\}$$
$$PQ = \{a(9), b(14), c(15), d(\infty), e(\infty), f(\infty), t(\infty)\}$$



Dijkstra's algorithm: an example

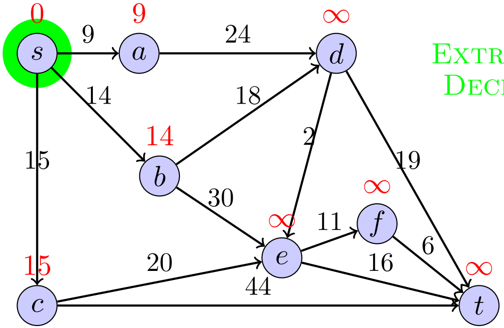
$$S = \{s\}$$
$$PQ = \{a(9), b(14), c(15), d(\infty), e(\infty), f(\infty), t(\infty)\}$$



EXTRACTMIN returns a

Dijkstra's algorithm: an example

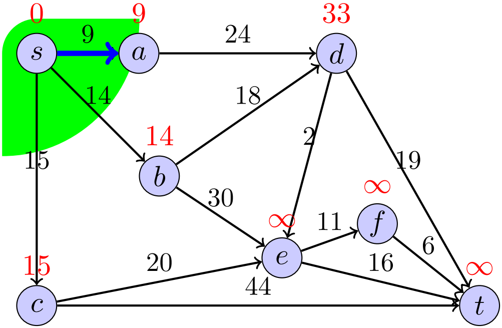
$$S = \{s\}$$
$$PQ = \{a(9), b(14), c(15), d(\infty), e(\infty), f(\infty), t(\infty)\}$$



$\textsc{ExtractMin}$ returns a
$\textsc{DecreaseKey}(d, 33)$

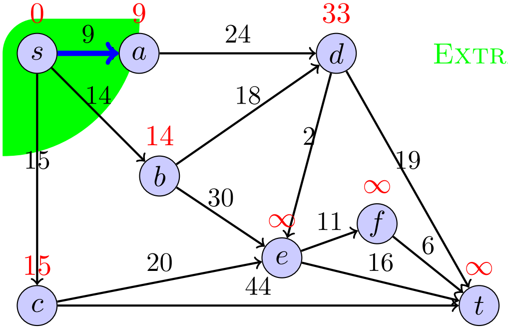
Dijkstra's algorithm: an example

$$S = \{s, a\}$$
$$PQ = \{b(14), c(15), d(33), e(\infty), f(\infty), t(\infty)\}$$



Dijkstra's algorithm: an example

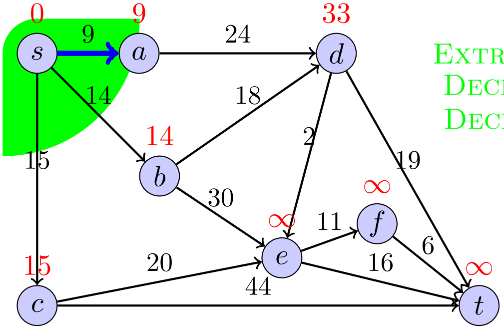
$$S = \{s, a\}$$
$$PQ = \{b(14), c(15), d(33), e(\infty), f(\infty), t(\infty)\}$$



ExtractMin returns b

Dijkstra's algorithm: an example

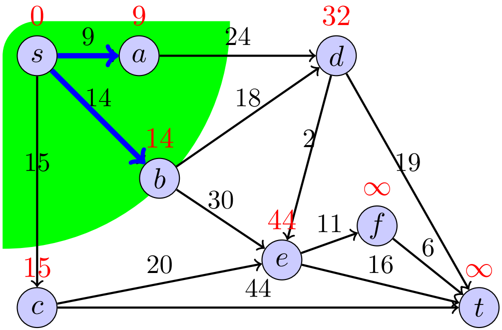
$$S = \{s, a\}$$
$$PQ = \{b(14), c(15), d(33), e(\infty), f(\infty), t(\infty)\}$$



ExtractMin returns b
$\text{DecreaseKey}(d, 32)$
$\text{DecreaseKey}(e, 44)$

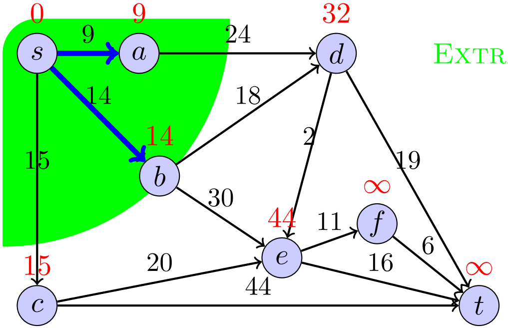
Dijkstra's algorithm: an example

$$S = \{s, a, b\}$$
$$PQ = \{c(15), d(32), e(44), f(\infty), t(\infty)\}$$



Dijkstra's algorithm: an example

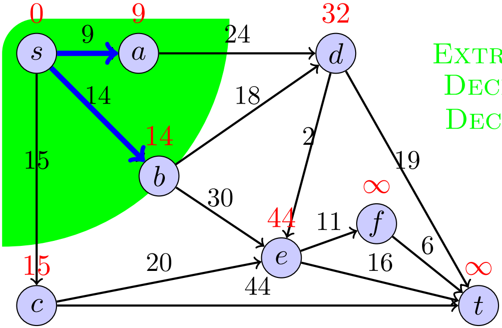
$$S = \{s, a, b\}$$
$$PQ = \{c(15), d(32), e(44), f(\infty), t(\infty)\}$$



EXTRACTMIN returns c

Dijkstra's algorithm: an example

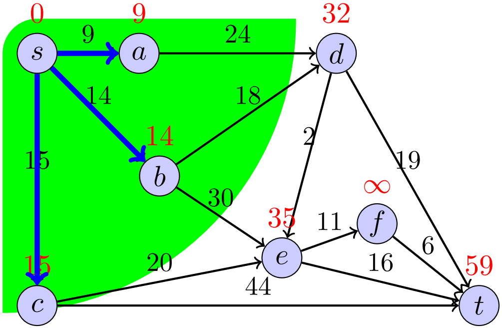
$$S = \{s, a, b\}$$
$$PQ = \{c(15), d(32), e(44), f(\infty), t(\infty)\}$$



ExtractMin returns c
DecreaseKey$(e, 35)$
DecreaseKey$(t, 59)$

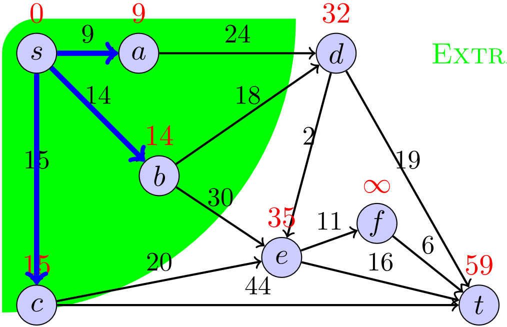
Dijkstra's algorithm: an example

$$S = \{s, a, b, c\}$$
$$PQ = \{d(32), e(35), t(59), f(\infty)\}$$



Dijkstra's algorithm: an example

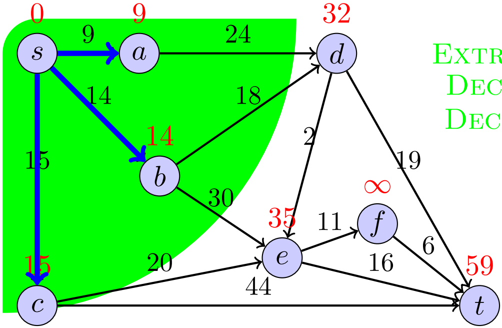
$$S = \{s, a, b, c\}$$
$$PQ = \{d(32), e(35), t(59), f(\infty)\}$$



EXTRACTMIN returns d

Dijkstra's algorithm: an example

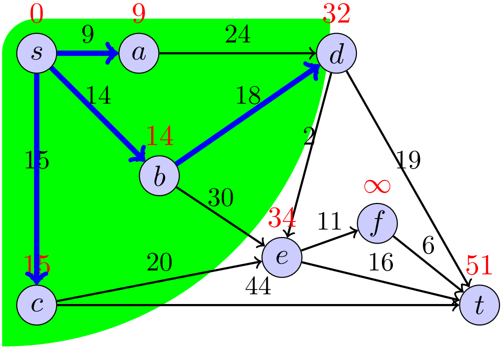
$$S = \{s, a, b, c\}$$
$$PQ = \{d(32), e(35), t(59), f(\infty)\}$$



EXTRACTMIN returns d
DECREASEKEY$(t, 51)$
DECREASEKEY$(e, 34)$

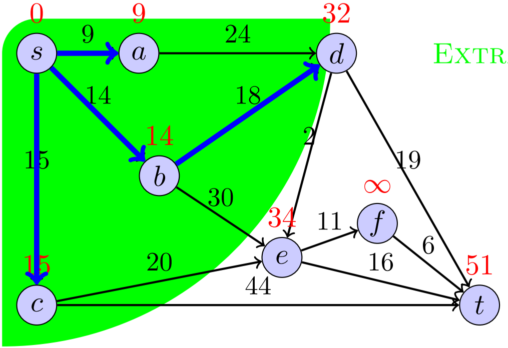
Dijkstra's algorithm: an example

$$S = \{s, a, b, c, d\}$$
$$PQ = \{e(34), t(51), f(\infty)\}$$



Dijkstra's algorithm: an example

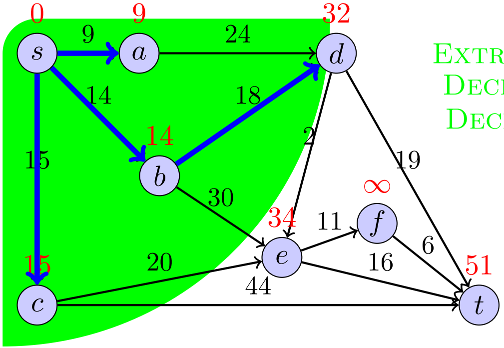
$$S = \{s, a, b, c, d\}$$
$$PQ = \{e(34), t(51), f(\infty)\}$$



ExtractMin returns e

Dijkstra's algorithm: an example

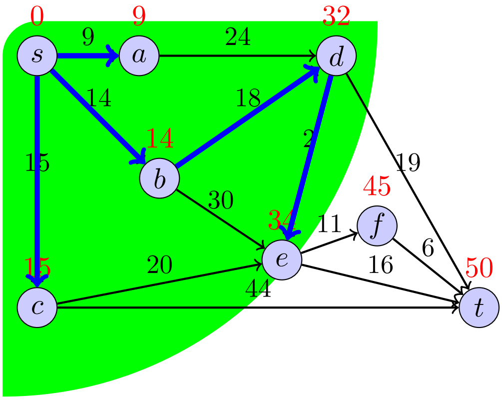
$$S = \{s, a, b, c, d\}$$
$$PQ = \{e(34), t(51), f(\infty)\}$$



$\textsc{ExtractMin}$ returns e
$\textsc{DecreaseKey}(f, 45)$
$\textsc{DecreaseKey}(t, 50)$

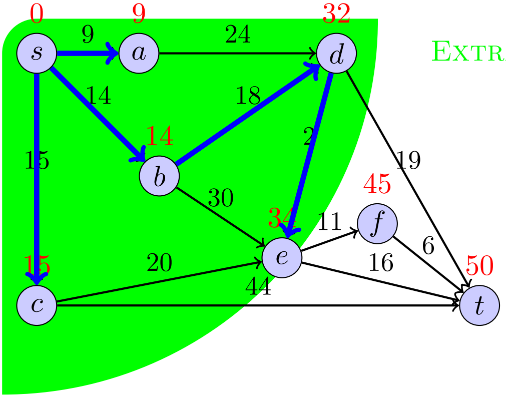
Dijkstra's algorithm: an example

$$S = \{s, a, b, c, d, e\}$$
$$PQ = \{f(45), t(50)\}$$



Dijkstra's algorithm: an example

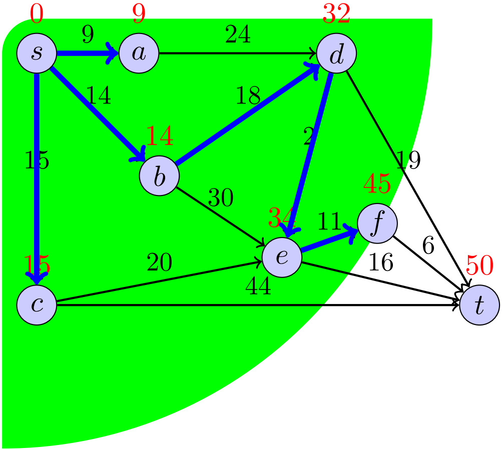
$S = \{s, a, b, c, d, e\}$
$PQ = \{f(45), t(50)\}$



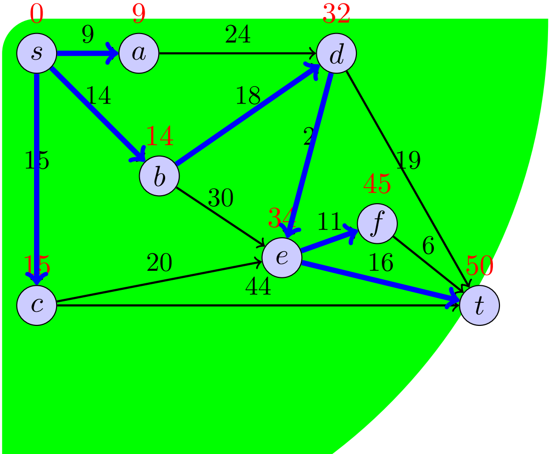
EXTRACTMIN returns f

Dijkstra's algorithm: an example

$$S = \{s, a, b, c, d, e, f\}$$
$$PQ = \{t(50)\}$$



$$S = \{s, a, b, c, d, e, f, t\}$$
$$PQ = \{\}$$



Time complexity of DIJKSTRA algorithm

Operation	Linked list	Binary heap	Binomial heap	Fibonacci heap
MAKEHEAP	1	1	1	1
INSERT	1	$\log n$	$\log n$	1
EXTRACTMIN	n	$\log n$	$\log n$	$\log n$
DECREASEKEY	1	$\log n$	$\log n$	1
DELETE	n	$\log n$	$\log n$	$\log n$
UNION	1	n	$\log n$	1
FINDMIN	n	1	$\log n$	1
DIJKSTRA	$O(n^2)$	$O(m \log n)$	$O(m \log n)$	$O(m + n \log n)$

DIJKSTRA algorithm: n INSERT, n EXTRACTMIN, and m DECREASEKEY.

Implementing priority queue: array or linked list

Implementing priority queue: unsorted array

- Unsorted array:

8	1	6	2	4			

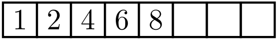
- Unsorted linked list:

$$\mathit{head} \to \boxed{8} \to \boxed{1} \to \boxed{6} \to \boxed{2} \to \boxed{4}$$

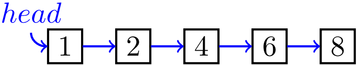
- Operations:
  - Insert: $O(1)$
  - ExtractMin: $O(n)$
- Note: a list containing n elements is too long to find the minimum efficiently.

Implementing priority queue: sorted array

- Sorted array:



- Sorted linked list:



- Operations:
  - INSERT: $O(n)$
  - EXTRACTMIN: $O(1)$
- Note: a list containing n elements is too long to maintain the order among elements.

Implementing priority queue: array or linked list

Operation	Linked List
INSERT	$O(1)$
EXTRACTMIN	$O(n)$
DECREASEKEY	$O(1)$
UNION	$O(1)$

Binary heap: from a linked list to a tree

Binary heap

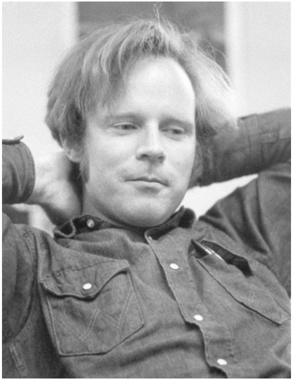
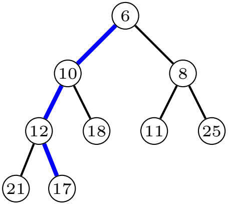


Figure 1: R. W. Floyd [1964]

Binary heap: a complete binary tree

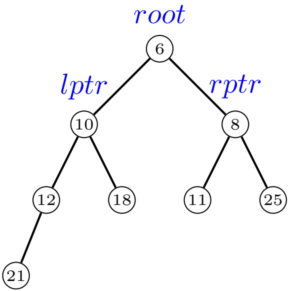
- Basic idea:
  - **loosing the structure**: Recall that the objective is to find the minimum. To achieve this objective, it is not necessary to sort all elements;
  - **but don't loose it too much**: we still need order between some elements;



- Binary heap: elements are stored in a **complete binary tree**, i.e., a tree that is perfectly balanced except for the bottom level. **Heap order** is required, i.e., any parent has a key smaller than his children;
- Advantage: any path has a short length of $O(\log_2 n)$ rather than n in linked list, making it efficient to maintain heap order.

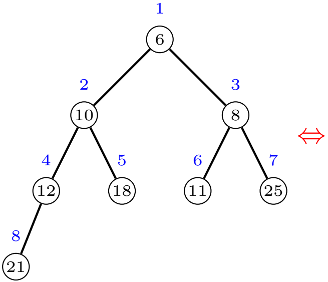
Binary heap: an explicit implementation

- **Pointer representation**: each node has pointers to its parent and two children;
- The following information are maintained:
  - the number of elements n;
  - the pointer to the root node;



- Note: the last node can be found in $O(\log n)$ time.

- **Array representation**: one-one correspondence between a binary tree and an array.
  - Binary tree $\Rightarrow$ array:
    - the indices starting from 1 for the sake of simplicity;
    - the indices record the order that the binary tree is traversed **level by level**.
  - Array $\Rightarrow$ binary tree:
    - the k-th item has two children located at $2k$ and $2k+1$;
    - the parent of the k-th item is located at $\lfloor \frac{k}{2} \rfloor$;



6	10	8	12	18	11	25	21			
1	2	3	4	5	6	7	8	9	10	11

Sorted array vs. binary heap

- Sorted array: an array containing n elements in an increasing order;

6	8	10	11	12	18	21	25			
1	2	3	4	5	6	7	8	9	10	11

- Binary heap: heap order means that only the order among nodes in short paths (length is less than $\log n$) are maintained. Note that some inverse pairs exist in the array.

6	10	8	12	18	11	25	21			
1	2	3	4	5	6	7	8	9	10	11

Binary heap: primitive and other operations

Primitive: exchanging nodes to restore heap order

- Primitive operation: when heap order is violated, i.e. a parent has a value larger than only one of its children, we simply exchange them to resolve the conflict.

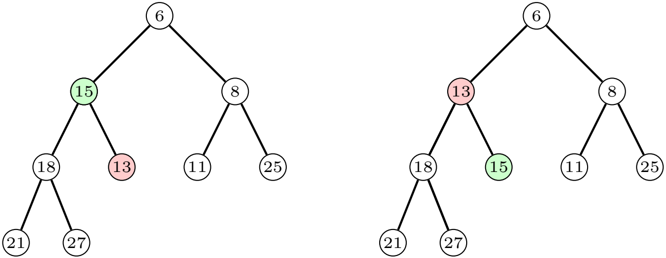


Figure 2: Heap order is violated: $15 > 13$. Exchange them to resolve the conflict.

Primitive: exchanging nodes to restore heap order

- Primitive operation: when heap order is violated, i.e. a parent has a value larger than both of its children, we exchange the parent with its smaller child to resolve the conflict.

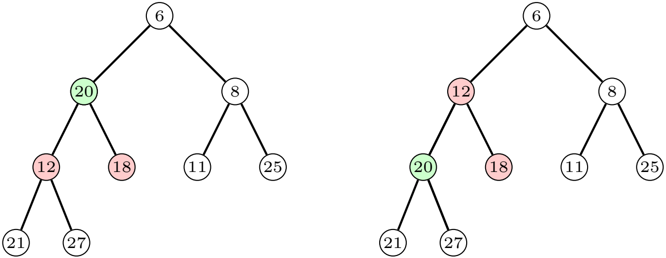
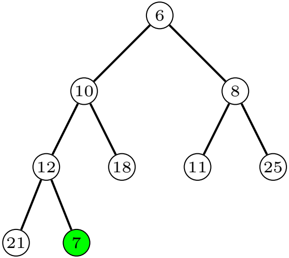
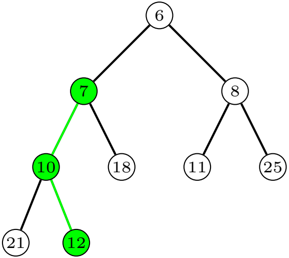


Figure 3: Heap order is violated: $20 > 12$, and $20 > 18$. Exchange 20 with its smaller child (12) to resolve the conflicts.

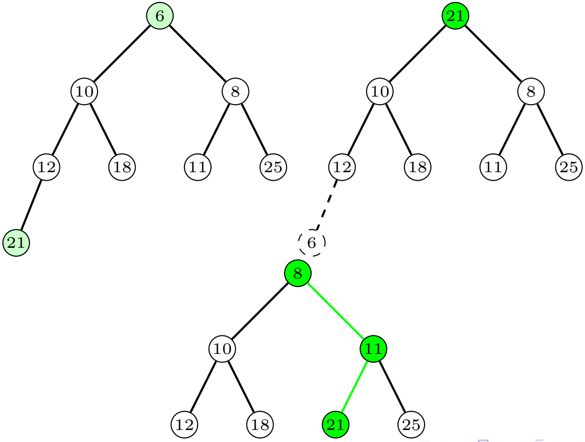
Binary heap: INSERT operation

- INSERT operation: the element is added as a new node at the end. Since the heap order might be violated, the node is repeatedly exchanged with its parent until heap order is restored.
- For example, INSERT(7):



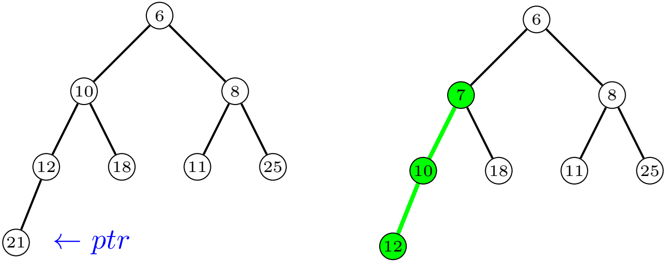


- ExtractMin operation: exchange element in root with the last node; repeatedly exchange the element in root with its **smaller child** until heap order is restored.
- For example, ExtractMin():



Binary heap: DecreaseKey operation

- DecreaseKey operation: given a handle to a node, repeatedly exchange the node with its parent until heap order is restored.
- For example, DecreaseKey(ptr, 7):



Binary heap: analysis

Theorem

In an **implicit** *binary heap, any sequence of m* INSERT, . DECREASEKEY, *and* EXTRACTMIN *operations with n* INSERT *operations takes $O(m \log n)$ time.*

Note:

- Each operation touches at most $\log n$ nodes on a path from the root to a leaf.

Theorem

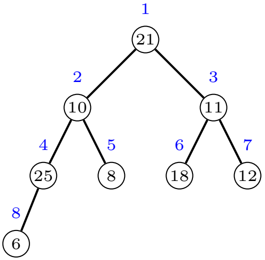
In an **explicit** *binary heap with n nodes, the* INSERT, . DECREASEKEY, *and* EXTRACTMIN *operations take $O(m \log n)$ time in the worst case.*

Note:

- If using array representation, a dynamic array expanding/contracting is needed. However, the total cost of array expanding/contracting is $O(n)$ (see TABLEINSERT).

Binary heap: heapify a set of items

- Question: Given a set of n elements, how to construct a binary heap containing them?
- Solutions:
  1. Simply INSERT the elements one by one. Takes $O(n \log n)$ time.
  2. Bottom-up heapifying. Takes $O(n)$ time.
     For $i = n$ to 1, we repeatedly exchange the element in node i with its smaller child until the subtree rooted at node i is heap-ordered.



21	10	11	25	8	18	12	6			
1	2	3	4	5	6	7	8	9	10	11

(see a demo)

Binary heap: heapify

Theorem

Given n elements, a binary heap can be constructed using $O(n)$ time.

Proof.

- There are at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of height h;
- It takes $O(h)$ time to sink a node of height h;
- The total time is:

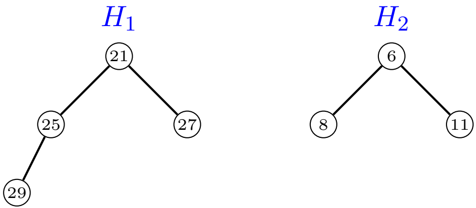
$$\begin{aligned}\sum_{h=0}^{\lfloor \log_2 n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil h &\leq \sum_{h=0}^{\lfloor \log_2 n \rfloor} n \frac{h}{2^h} \\ &\leq 2n\end{aligned}$$

$\square$

Implementing priority queue: binary heap

Operation	Linked List	Binary Heap
Insert	$O(1)$	$O(\log n)$
ExtractMin	$O(n)$	$O(\log n)$
DecreaseKey	$O(1)$	$O(\log n)$
Union	$O(1)$	$O(n)$

- UNION operation: Given two binary heaps H_1 and H_2, to merge them into one binary heap.



- $O(n)$ time is needed if using heapify.
- Question: Is there a quicker way to union two heaps?

Binomial heap: using multiple trees rather than a single tree to support efficient UNION operation

Binomial heap

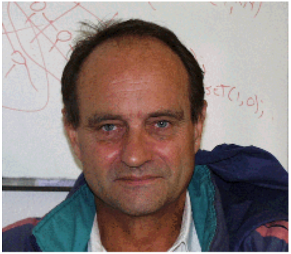
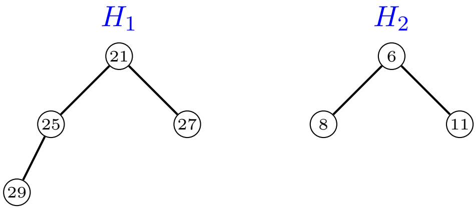


Figure 4: Jean Vuillenmin [1978]

Binomial heap: efficient UNION

- Basic idea:
  - **loosing the structure**: if **multiple trees** are allowed to represent a heap, UNION can be efficiently implemented via simply putting trees together.
  - **but don't loose it too much**: there should not be too many trees; otherwise, it will take a long time to find the minimum among all root nodes.

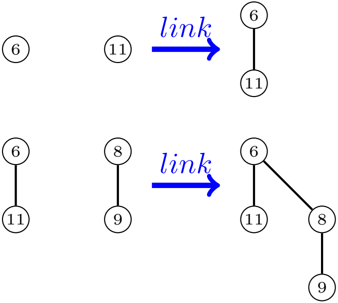


- EXTRACTMIN: simply finding the minimum element of the root nodes. Note that a root node holds the minimum of the tree due to the heap order.

Why we can't loose the structure too much?

- An extreme case of multiple trees: each node is itself a tree. Then it will take $O(n)$ time to find the minimum.

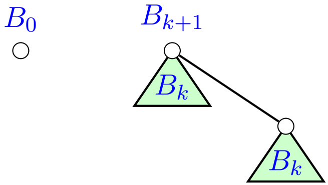
- Solution: **consolidating**, i.e., two trees (with the same size) are merged into one — the larger root is linked to the smaller one to keep the heap order. Note that after consolidating, at most $\log n$ trees will be left.



Binomial tree

Definition (Binomial tree)

The binomial tree is defined recursively: a single node is itself a B_0 tree, and two B_k trees are linked into a B_{k+1} tree.

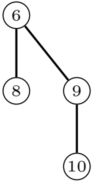


Binomial tree examples: B_0, B_1, B_2

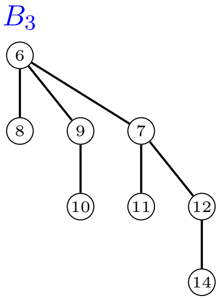
B_0

B_1

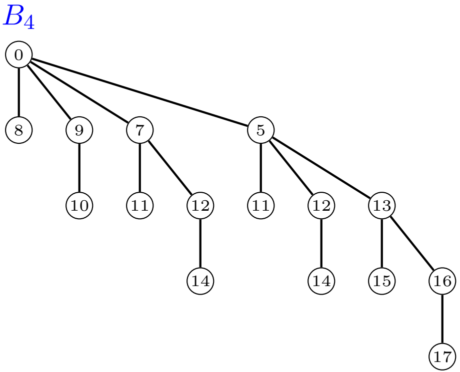
B_2



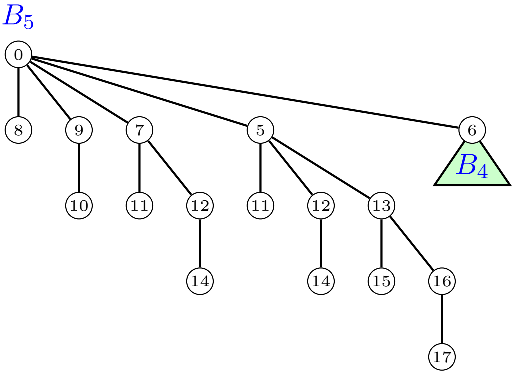
Binomial tree example: B_3



Binomial tree example: B_4



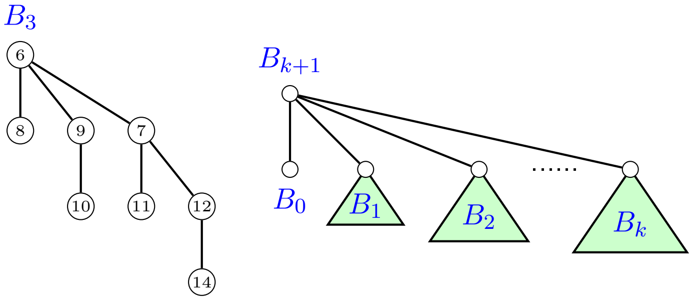
Binomial tree example: B_5



Binomial tree: property

Properties:

1. $|B_k| = 2^k$.
2. $height(B_k) = k$.
3. $degree(B_k) = k$.
4. The i-th child of a node has a degree of $i - 1$.
5. The deletion of the root yields trees B_0, B_1, ..., B_{k-1}.
6. Binomial tree is named after the fact that the node number of all levels are binomial coefficients.



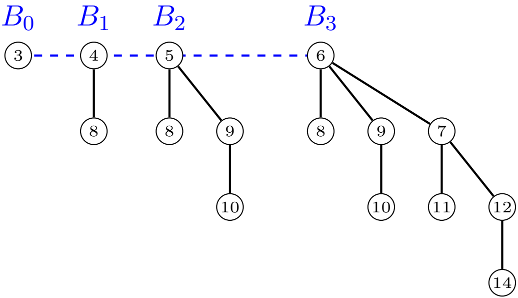
Binomial heap: a forest

Definition (Binomial forest)

A binomial heap is a collection of several binomial trees:

- Each tree is heap ordered;
- There is either 0 or 1 B_k for any k.

- Example:



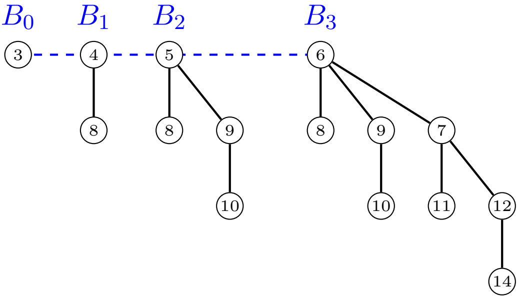
- Note that the roots are organized using doubly-linked circular list, and the minimum of them is recorded using a pointer.

Binomial heap: properties

Properties:

1. A binomial heap with n nodes contains the binomial tree B_i iff $b_i = 1$, where $b_k b_{k-1} \ldots b_1 b_0$ is binary representation of n.
2. It has at most $\lfloor \log_2 n \rfloor + 1$ trees.
3. Its height is at most $\lfloor \log_2 n \rfloor$.

Thus, it takes $O(\log n)$ time to find the minimum element via checking the roots.



UNION is efficient: example 1

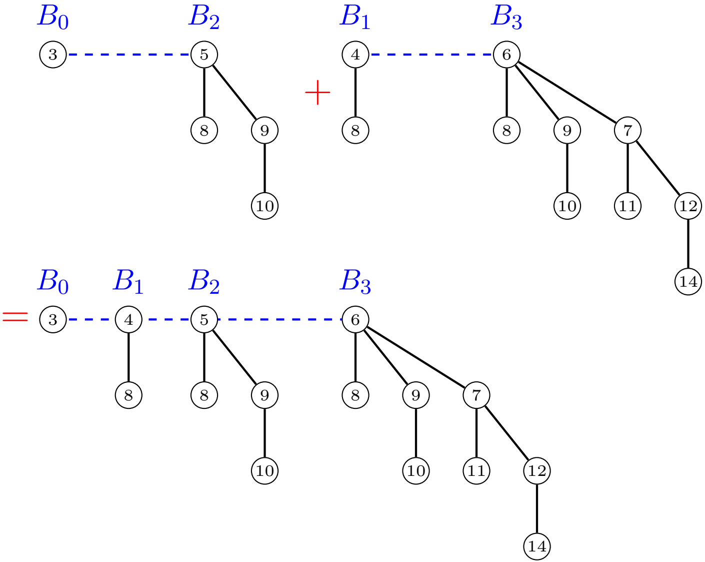


Figure 5: An easy case: no consolidating is needed

UNION is efficient: example 2 I

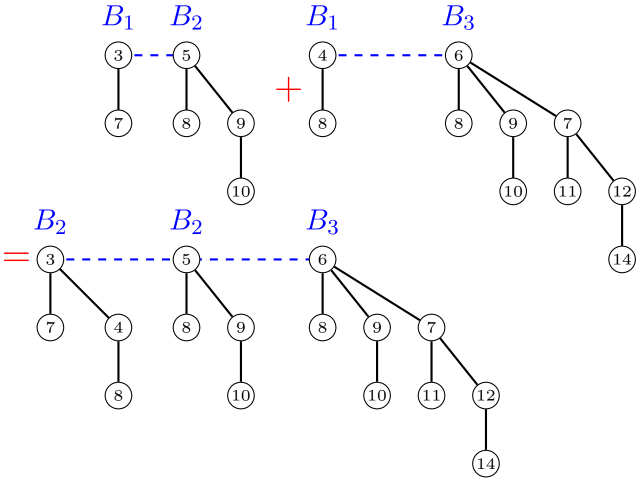


Figure 6: Consolidating two B_1 trees into a B_2 tree

UNION is efficient: example 2 II

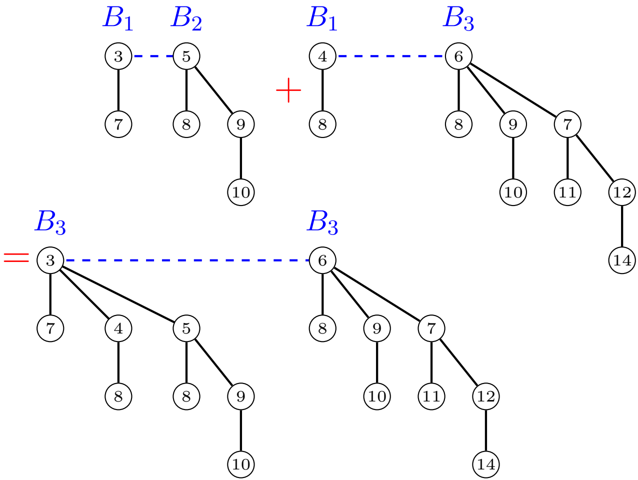
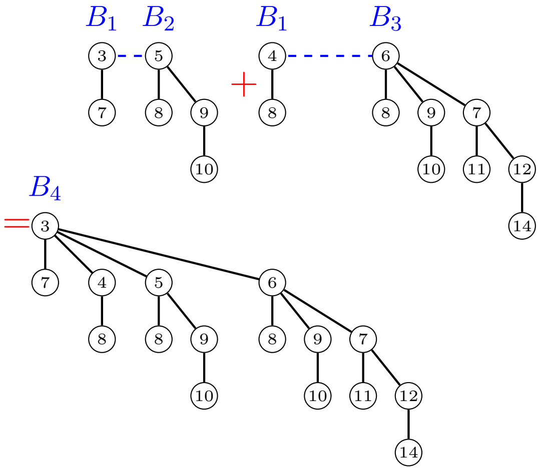


Figure 7: Consolidating two B_2 trees into a B_3 tree

UNION is efficient: example 2 III



Time complexity: $O(\log n)$ since there are at most $O(\log n)$ trees.

Binomial heap: INSERT operation

INSERT(x)

```
Create a B_0 tree for x;
Change the pointer to the minimum root node if necessary;
while there are two B_k trees for some k do
    Link them together into one B_{k+1} tree;
    Change the pointer to the minimum root node if necessary;
end while
```

INSERT operation: an example

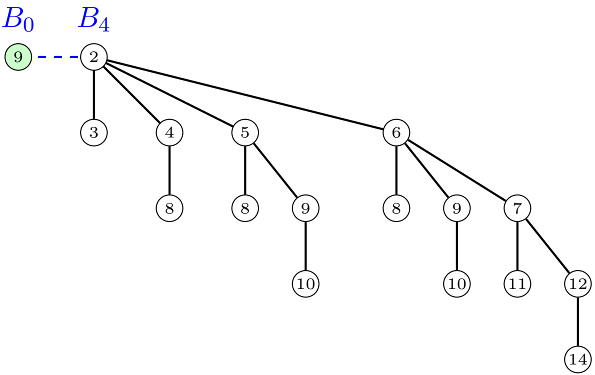


Figure 8: An easy case: no consolidating is needed

INSERT operation: example 2 I

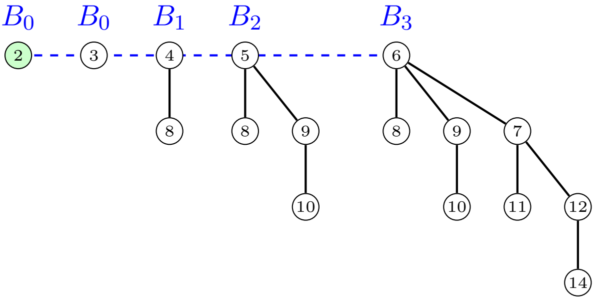


Figure 9: Consolidating two B_0

INSERT operation: example 2 II

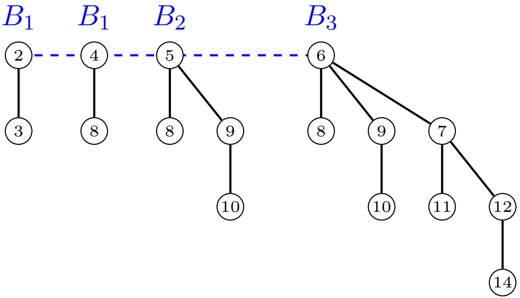


Figure 10: Consolidating two B_1

INSERT operation: example 2 III

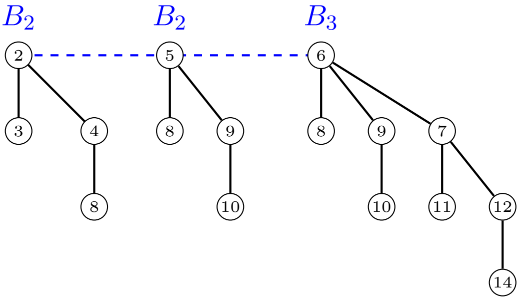


Figure 11: Consolidating two B_2

INSERT operation: example 2 IV

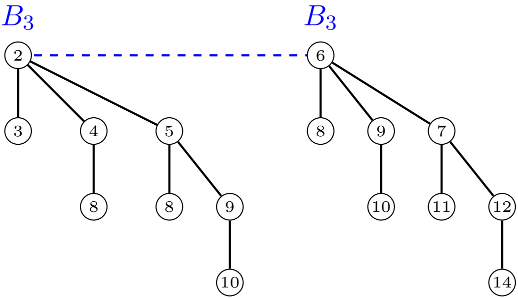
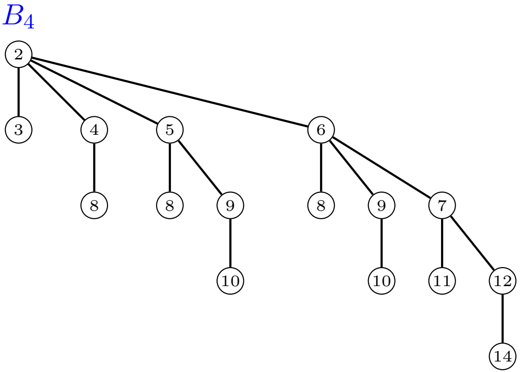


Figure 12: Consolidating two B_3



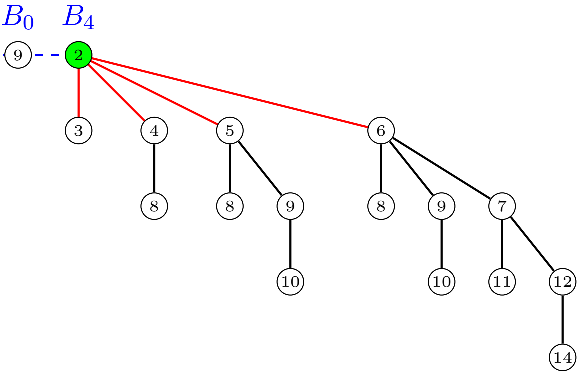
Time complexity: $O(\log n)$ (worst case) since there are at most $\log n$ trees.

Binomial heap: EXTRACTMIN operation

EXTRACTMIN()

1: Remove the min node, and insert its children into the root list;
2: Change the pointer to the minimum root node if necessary;
3: **while** there are two B_k trees for some k **do**
4: Link them together into one B_{k+1} tree;
5: Change the pointer to the minimum root node if necessary;
6: **end while**

ExtractMin operation: an example I



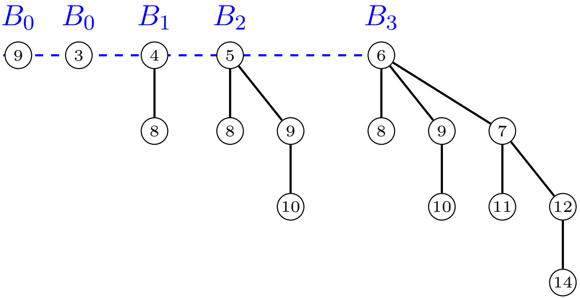


Figure 13: The four children become trees

ExtractMin operation: an example III

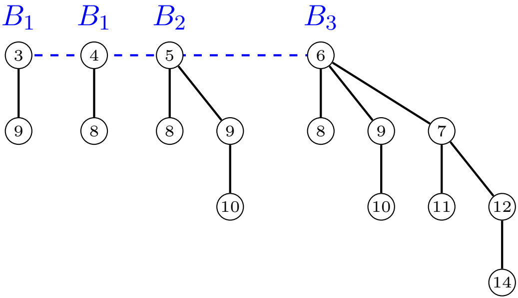


Figure 14: Consolidating two B_1 trees

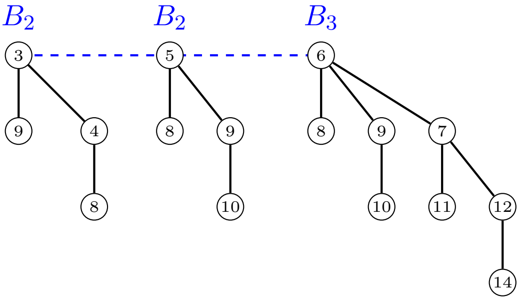


Figure 15: Consolidating two B_2 trees

ExtractMin operation: an example V

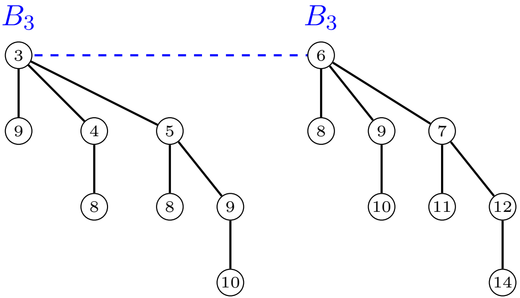
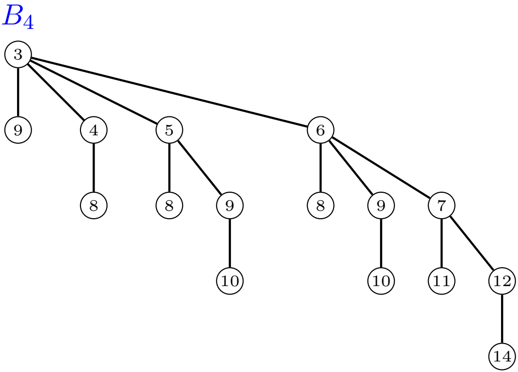


Figure 16: Consolidating two B_2 trees



Time complexity: $O(\log n)$

Implementing priority queue: Binomial heap

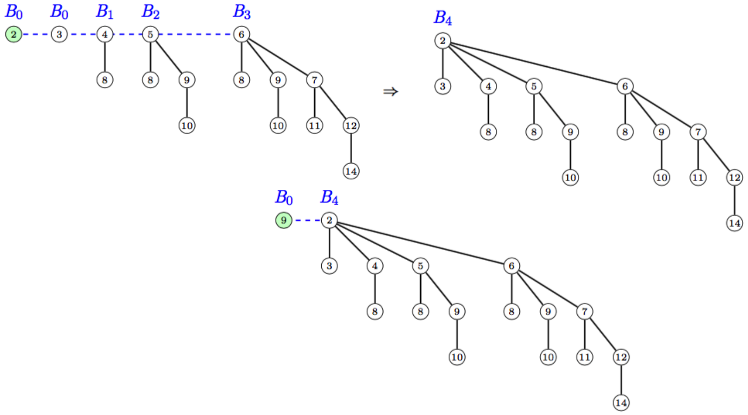
Operation	Linked List	Binary Heap	Binomial Heap
Insert	$O(1)$	$O(\log n)$	$O(\log n)$
ExtractMin	$O(n)$	$O(\log n)$	$O(\log n)$
DecreaseKey	$O(1)$	$O(\log n)$	$O(\log n)$
Union	$O(1)$	$O(n)$	$O(\log n)$

Binomial heap: more accurate analysis using the amortized technique

Amortized analysis of INSERT

Motivation:

- If an INSERT takes a long time (say $\log n$), the subsequent INSERT operations shouldn't take long!



- Thus, it will be more accurate to examine a sequence of operations rather than each operation individually.

Amortized analysis of INSERT operation

INSERT(x)

```
Create a B_0 tree for x;
Change the pointer to the minimum root node if necessary;
while there are two B_k trees for some k do
    Link them together into one B_{k+1} tree;
    Change the pointer to the minimum root node if necessary;
end while
```

Analysis:

- A single INSERT operation takes time $1 + w$, where $w = \#\texttt{WHILE}$.
- For the sake of calculating the total running time of a sequence of operations, we represent the running time of a single operation as decrease of a potential function.
- Consider a quantity $\Phi = \#trees$ (called potential function). The changes of Φ during an operation are:
  - Φ increase: 1.
  - Φ decrease: w.
- Thus the running time of INSERT can be rewritten in terms of Φ as $1 + w = 1+$ decrease in Φ. Note that this representation makes it convenient to sum running time of a sequence of INSERT operations.

Amortized analysis of ExtractMin

ExtractMin()

```
Remove the min node, and insert its children to the root list;
Change the pointer to the minimum root node if necessary;
while there are two B_k trees for some k do
    Link them together into one B_{k+1} tree;
    Change the pointer to the minimum root node if necessary;
end while
```

Analysis:

- A single ExtractMin operation takes $d + w$ time, where d denotes degree of the removed root node, and $w = \#\texttt{WHILE}$.
- For the sake of calculating the total running time of a sequence of operations, we represent the running time of a single operation as decrease of a potential function.
- Consider a potential function $\Phi = \#trees$. The changes during an operation are:
    - Φ increase: d.
    - Φ decrease: w.
- Similarly, the running time is rewritten in terms of Φ as $d + w = d+$ decrease in $\#trees$. Note that $d \leq \log n$.

- Let's consider any sequence of n $\textsc{Insert}$ and m $\textsc{ExtractMin}$ operations.
- The total running time is at most $n + m \log n +$ total decrease in $\#trees$.
- Note: total decrease in $\#trees \leq$ total increase in $\#trees$ (why?), which is at most $n + m \log n$.
- Thus the total time is at most $2n + 2m \log n$.
- We say $\textsc{Insert}$ takes $O(1)$ amortized time, and $\textsc{ExtractMin}$ takes $O(\log n)$ amortized time.

Definition (Amortized time)

For any sequence of n_1 operation 1, n_2 operation 2..., if the total time is $O(n_1T_1 + n_2T_2...)$, we say that operation 1 takes T_1 amortized time, operation 2 takes T_2 amortized time ....

Intuition of the amortized analysis

- The actual running time of an INSERT operation is $1 + w$. A large w means that the INSERT operation takes a long time. Note that the w time was spent on "decreasing trees"; thus, if the w time was amortized over the operations "creating trees", the "amortized time" of INSERT operation will be only $O(1)$.
- The actual running time of an EXTRACTMIN operation is at most $\log n + w$. Note that at most $\log n$ new trees are created during an EXTRACTMIN operation; thus, the amortized time is still $O(\log n)$ even if some costs have been amortized to it from other operations due to "tree creating".

Implementing priority queue: Binomial heap

Operation	Linked List	Binary Heap	Binomial Heap	Binomial Heap*
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
EXTRACTMIN	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECREASEKEY	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
UNION	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$

*amortized cost

Binomial heap: DecreaseKey operation

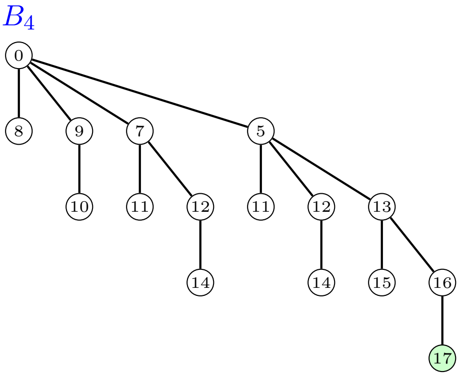


Figure 17: DecreaseKey: 17 to 1

- Time: $O(\log n)$ since in the worst case, we need to perform node exchanging up to the root.
- Question: is there a quicker way for decrease key?

Fibonacci heap: an efficient implementation of DecreaseKey via simply cutting an edge rather than exchanging nodes

Figure 18: Robert Tarjan [1986]

- Basic idea:
  - **loosing the structure**: When heap order is violated, a simple solution is to "cut off a node, and insert it into the root list".
  - **but don't loose it too much**: the "cutting off" operation makes a tree not "binomial" any more; however, it should not deviate from a binomial tree too much. A technique to achieve this objective is allowing any non-root node to lose "at most one child".

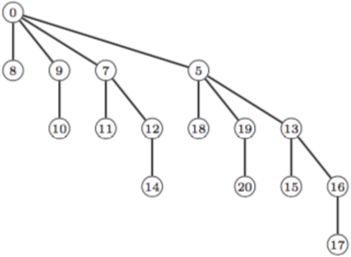


Figure 19: Heap order is violated when DESCREASEKEY 13 to 2. However, the heap order can be easily restored via "cutting off" the node, and inserting it into the root list

Fibonacci heap: DecreaseKey

DecreaseKey(v, x)

$key(v) = x$;
if heap order is violated **then**
    $u = v's$ parent;
    Cut subtree rooted at node v, and insert it into the root list;
    Change the pointer to the minimum root node if necessary;
    while u is marked **do**
        Cut subtree rooted at node u, and insert it into the root list;
        Change the pointer to the minimum root node if necessary;
        Unmark u;
        $u = u's$ parent;
    end while
    Mark u;
end if

DecreaseKey: an example I

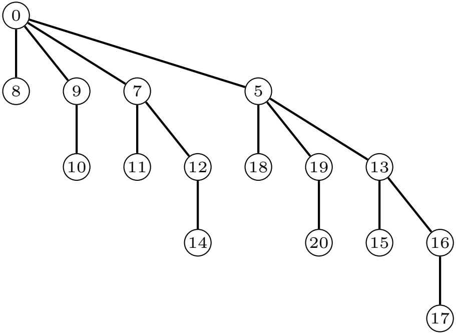


Figure 20: A Fibonacci heap. To DecreaseKey: 19 to 3.

DecreaseKey: an example II

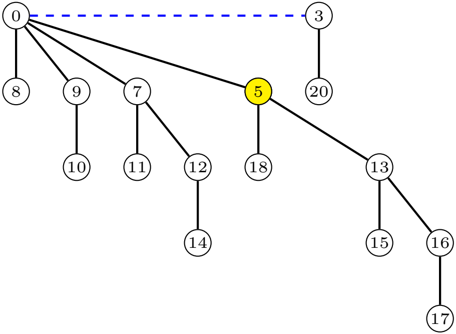


Figure 21: After DecreaseKey: 19 to 3. To DecreaseKey: 15 to 2.

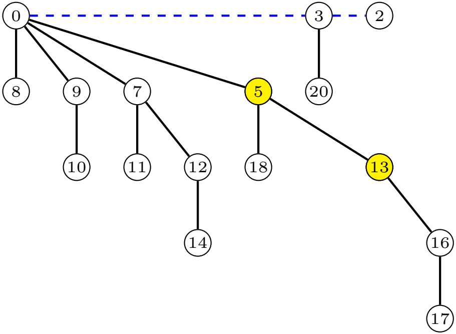


Figure 22: After DECREASEKEY: 15 to 2. To DECREASEKEY: 12 to 8.

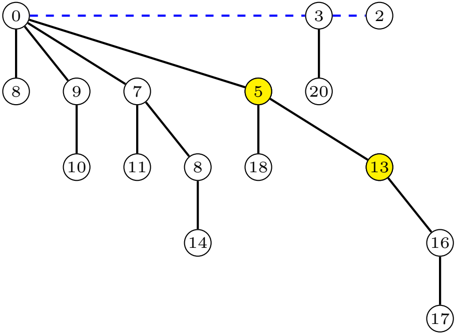


Figure 23: After DecreaseKey: 12 to 8. To DecreaseKey: 14 to 1.

DecreaseKey: an example V

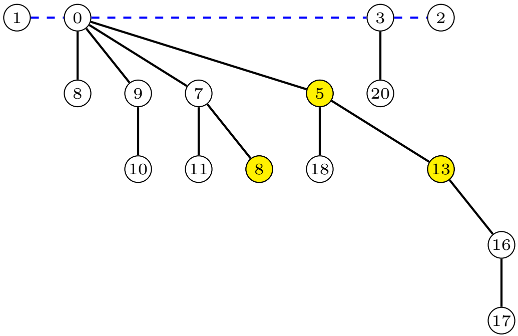


Figure 24: After DecreaseKey: 14 to 1. To DecreaseKey: 16 to 9.

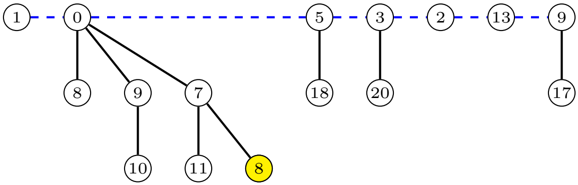


Figure 25: After DecreaseKey: 16 to 9

INSERT(x)

1: Create a tree for x, and insert it into the root list;
2: Change the pointer to the minimum root node if necessary;

Note: **Being lazy!** Consolidating trees when extracting minimum.

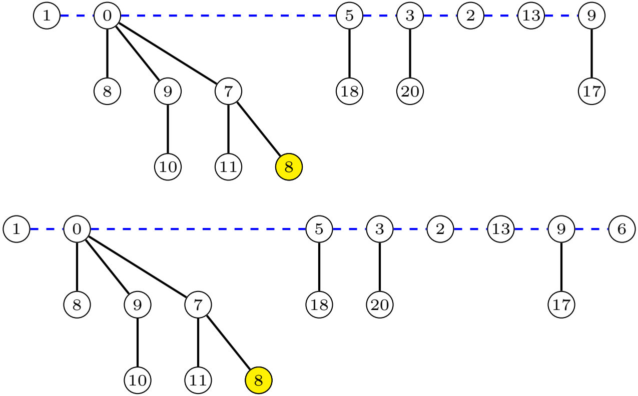


Figure 26: INSERT(6): creating a new tree, and insert it into the root list

Fibonacci heap: ExtractMin

ExtractMin()

```
Remove the min node, and insert its children into the root list;
Change the pointer to the minimum root node if necessary;
while there are two roots u and v of the same degree do
    Consolidate the two trees together;
    Change the pointer to the minimum root node if necessary;
end while
```

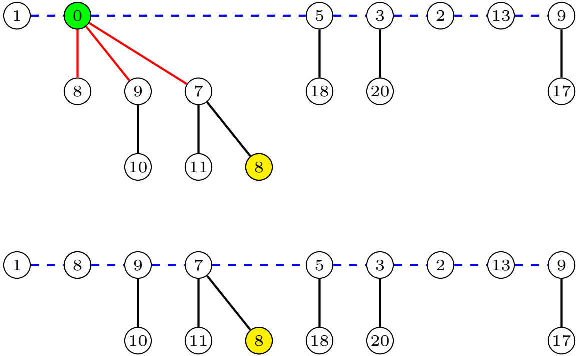


Figure 27: ExtractMin: removing the min node, and adding 3 trees

ExtractMin: an example II

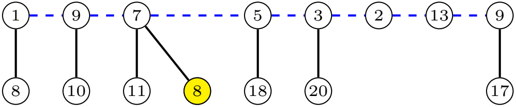


Figure 28: ExtractMin: after consolidating two trees rooted at node 1 and 8

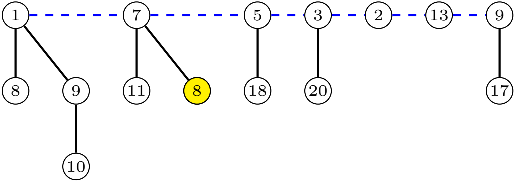


Figure 29: ExtractMin: after consolidating two trees rooted at node 1 and 9

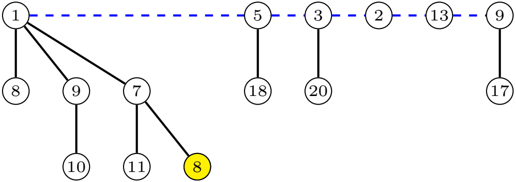


Figure 30: ExtractMin: after consolidating two trees rooted at node 1 and 7

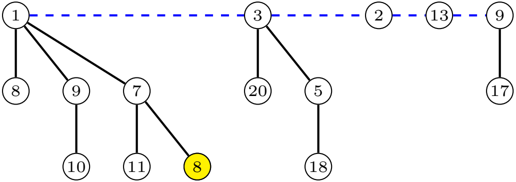


Figure 31: ExtractMin: after consolidating two trees rooted at node 3 and 5

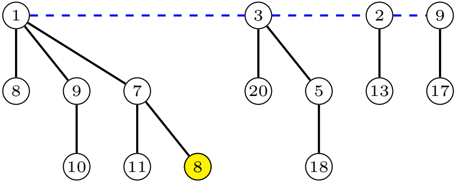


Figure 32: EXTRACTMIN: after consolidating two trees rooted at node 2 and 13

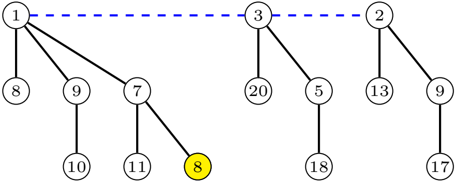


Figure 33: EXTRACTMIN: after consolidating two trees rooted at node 2 and 9

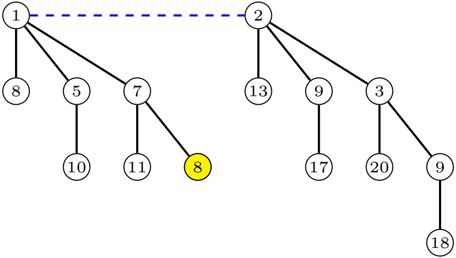


Figure 34: ExtractMin: after consolidating two trees rooted at node 2 and 3

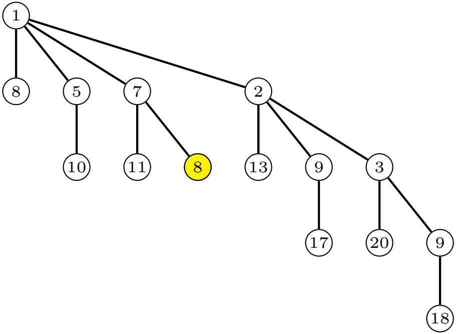


Figure 35: ExtractMin: after consolidating two trees rooted at node 1 and 2

Fibonacci heap: an amortized analysis

DecreaseKey(v, x)

1. $key(v) = x$;
2. **if** heap order is violated **then**
3. $u = v's$ parent;
4. Cut subtree rooted at node v, and insert it into the root list;
5. Change the pointer to the minimum root node if necessary;
6. **while** u is marked **do**
7. Cut subtree rooted at node u, and insert it into the root list;
8. Change the pointer to the minimum root node if necessary;
9. Unmark u;
10. $u = u's$ parent;
11. **end while**
12. Mark u;
13. **end if**

Analysis:

- The actual running time of a single operation is $1 + w$, where $w = \#\mathtt{WHILE}$.
- For the sake of calculating the total running time of a sequence of operations, we represent the running time of a single operation as decrease of a potential function.
- Consider a potential function $\Phi = \#trees + 2\#marks$. The changes of Φ during an operation are:
  - Φ increase: $1 + 2 = 3$.
  - Φ decrease: $(-1 + 2 * 1) * w = w$.
- Thus we can rewrite the running time in terms of Φ as $1 + w = 1 + \Phi$ decrease.

Intuition: a large w means that DecreaseKey takes a long time; however, if we can "amortize" w over other operations, a DecreaseKey operation takes only $O(1)$ "amortized time".

Fibonacci heap: ExtractMin

ExtractMin()

```
Remove the min node, and insert its children into the root list;
Change the pointer to the minimum root node if necessary;
while there are two roots u and v of the same degree do
    Consolidate the two trees together;
    Change the pointer to the minimum root node if necessary;
end while
```

Analysis:

- The actual running time of a single operation is $d + w$, where d denotes degree of the removed node, and $w = \#\texttt{WHILE}$.
- For the sake of calculating the total running time of a sequence of operations, we represent the running time of a single operation as decrease of a potential function.
- Consider a potential function $\Phi = \#trees + 2\#marks$. The changes of Φ during an operation are:
  - Φ increase: d.
  - Φ decrease: w.
- Thus the running time can be rewritten in terms of Φ as $d + w = d +$ decrease in Φ.

Note: $d \leq d_{max}$, where d_{max} denotes the maximum root node degree.

Fibonacci heap: INSERT

INSERT(x)

1: Create a tree for x, and insert it into the root list;

2: Change the pointer to the minimum root node if necessary;

Analysis:

- The actual running time is 1, and the changes of Φ during this operation are:
  - Φ increase: 1.
  - Φ decrease: 0.

Note:

- Recall that a binomial heap consolidates trees in both INSERT and EXTRACTMIN operations.
- In contrast, the Fibonacci heap adopts the strategy of **"being lazy"** — tree consolidating is removed from INSERT operation for the sake of efficiency, and there is no tree consolidating until an EXTRACTMIN operation.

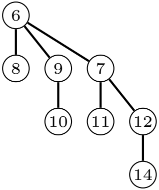
Fibonacci heap: amortized analysis

- Consider any sequence of n $\textsc{Insert}$, m $\textsc{ExtractMin}$, and r $\textsc{DecreaseKey}$ operations.
- The total running time is at most: $n + md_{max} + r +$ total decrease in Φ.
- Note: total decrease in $\Phi \leq$ total increase in $\Phi =$ $n + md_{max} + 3r$.
- Thus the total running time is at most: $n + md_{max} + r + n + md_{max} + 3r = 2n + 2md_{max} + 4r$.
- Thus $\textsc{Insert}$ takes $O(1)$ amortized time, $\textsc{DecreaseKey}$ takes $O(1)$ amortized time, and $\textsc{ExtractMin}$ takes $O(d_{max})$ amortized time.
- In fact, $\textsc{ExtractMin}$ takes $O(\log n)$ amortized time since d_{max} can be upper-bounded by $\log n$ (why?).

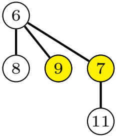
Fibonacci heap: bounding d_{max}

Fibonacci heap: bounding d_{max}

- Recall that for a binomial tree having n nodes, the root degree d is **exactly** $\log_2 n$, i.e. $d = \log_2 n$.



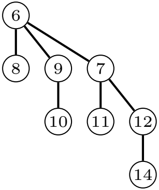
- In contrast, a tree in a Fibonacci heap might have several subtrees cutting off, leading to $d \geq \log_2 n$.



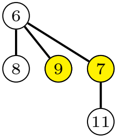
- However, the "marking technique" guarantees that any node can lose at most one child, thus limiting the deviation from the original binomial tree, i.e. $\log_\phi n \geq d \geq \log_2 n$, where $\phi = \frac{1+\sqrt{5}}{2} = 1.618$.

Fibonacci heap: a property of node degree

- Recall that for a binomial tree, the i-th child of each node has a degree of exactly $i - 1$.

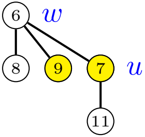


- For a tree in a Fibonacci heap , we will show that the i-th child of each node has degree $\geq i - 2$.



Lemma

For any node in a Fibonacci heap, the i-th child has a degree $\geq i - 2$.



Proof.

- Suppose u is the **current** i-th child of w;
- If w is not a root node, it has at most 1 child lost; otherwise, it might have multiple children lost;
- Consider the time when u is linked to w. At that time, $degree(w) \geq i - 1$, so $degree(u) = degree(w) \geq i - 1$;
- Subsequently, $degree(u)$ decreases by at most 1 (Otherwise, u will be cut off and no longer a child of w).
- Thus, $degree(u) \geq i - 2$.

- Let F_k be **the smallest tree** with root degree of k, and for any node of F_k, the i-th child has degree $\geq i - 2$;

B_1

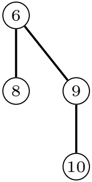
F_0

Figure 36: $|B_1| = 2^1$ and $|F_0| = 1 \geq \phi^0$

Example: B_2 versus F_1

- Let F_k be the smallest tree with root degree of k, and for any node of F_k, the i-th child has degree $\geq i - 2$;

B_2



F_1

Figure 37: $|B_2| = 2^2$ and $|F_1| = 2 \geq \phi^1$

Example: B_3 versus F_2

- Let F_k be the smallest tree with root degree of k, and for any node of F_k, the i-th child has degree $\geq i - 2$;

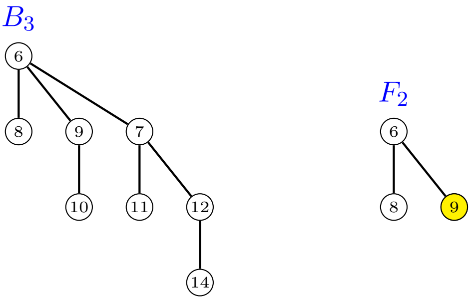


Figure 38: $|B_3| = 2^3$ and $|F_2| = 3 \geq \phi^2$

Example: B_4 versus F_3

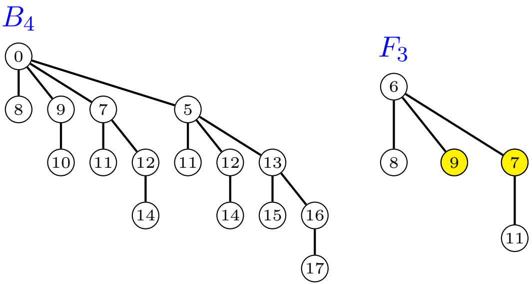


Figure 39: $|B_4| = 2^4$ and $|F_3| = 5 \geq \phi^3$

Example: B_5 versus F_4

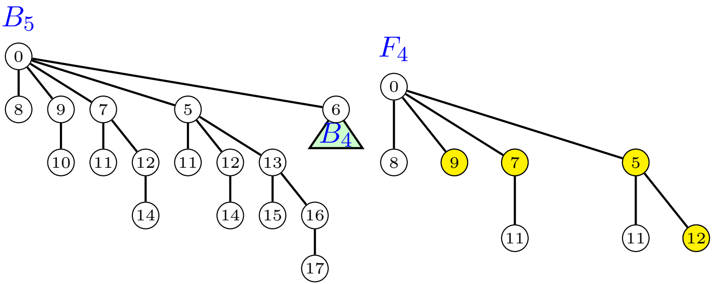
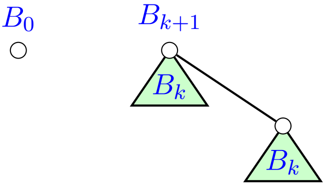


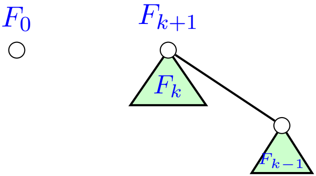
Figure 40: $|B_5| = 2^5$ and $|F_4| = 8 \geq \phi^4$

General case of trees in Fibonacci heap

- Recall that a binomial tree B_{k+1} is a combination of two B_k trees.



- In contrast, F_{k+1} is the combination of an F_k tree and an F_{k-1} tree.



- We will show that though F_k is smaller than B_k, the difference is not too much. In fact, $|F_k| \geq 1.618^k$.

Fibonacci numbers and Fibonacci heap

Definition (Fibonacci numbers)

The Fibonacci sequence is $0, 1, 1, 2, 3, 5, 8, 13, 21, 34 \ldots$. It can be defined by the recursion relation: $f_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ f_{k-1} + f_{k-2} & \text{if } k \geq 2 \end{cases}$

- Recall that $f_{k+2} \geq \phi^k$, where $\phi = \frac{1+\sqrt{5}}{2} = 1.618\ldots$.
- Note that $|F_k| = f_{k+2}$, say $|F_0| = f_2 = 1$, $|F_1| = f_3 = 2$, $|F_2| = f_4 = 3$.
- Consider a Fibonacci heap H having n nodes. Let T denote a tree in H with root degree d.
- We have $n \geq |T| \geq |F_d| = f_{d+2} \geq \phi^d$.
- Thus $d = O(\log_\phi n) = O(\log n)$. So, $d_{max} = O(\log n)$.

Therefore, $\textsc{ExtractMin}$ operation takes $O(\log n)$ amortized time.

Implementing priority queue: Fibonacci heap

Operation	Linked List	Binary Heap	Binomial Heap	Binomial Heap*	Fibonacci Heap*
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
EXTRACTMIN	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECREASEKEY	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$
UNION	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$	$O(1)$

*amortized cost

Time complexity of DIJKSTRA algorithm

Operation	Linked list	Binary heap	Binomial heap	Fibonacci heap
MAKEHEAP	1	1	1	1
INSERT	1	$\log n$	$\log n$	1
EXTRACTMIN	n	$\log n$	$\log n$	$\log n$
DECREASEKEY	1	$\log n$	$\log n$	1
DELETE	n	$\log n$	$\log n$	$\log n$
UNION	1	n	$\log n$	1
FINDMIN	n	1	$\log n$	1
DIJKSTRA	$O(n^2)$	$O(m \log n)$	$O(m \log n)$	$O(m + n \log n)$

DIJKSTRA algorithm: n INSERT, n EXTRACTMIN, and m DECREASEKEY.