Priority Queues, Heaps, and Heapsort

CSE 2320 – Algorithms and Data Structures
Vassilis Athitsos
University of Texas at Arlington

Priority Queues

- So far we have seen sorting methods that works in **batch mode**:
  - They are given all the items at once
  - They sort the items.
  - Done!
- Another case of interest is **online** methods, that deal with data that change.
- Goal: support (efficiently):
  - Insertion of a new element.
  - Deletion of the max element.
  - Initialization (organizing an initial set of data).
- The abstract data type that supports these operations is called **priority queue**.

Priority Queues - Applications

- Scheduling:
  - Flight take-offs and landings.
  - Programs getting executed on a computer.
  - Real-time requests for information on a database system.
  - Computer simulations and games, to schedule a sequence of events.
- Waiting lists:
  - Students getting admitted to college.
  - Patients getting admitted to a hospital.
- Lots more…

Priority Queues and Sorting

- Priority queues support:
  - Insertion of a new element.
  - Deletion of the max element.
  - Initialization (organizing an initial set of data).
- These operations support applications that batch methods, like quicksort, mergesort, do not support.
- However, these operations can also support sorting:
- Given items to sort:
  - Initialize a priority queue that contains those items.
  - Initialize result to empty list.
  - While the priority queue is not empty:
    - Remove max element from queue, add it to beginning of result.
- We will see an implementation (heapsort) of this algorithm that takes $\Theta(N \lg N)$ time.

Naïve Implementation Using Arrays

- Initialization:
  - Given N data, just store them on an array.
  - Time: Θ(???)
- Insertion of a new item:
  - (Assumption: the array has enough memory.)
  - Store the item at the end of the array.
  - Time: Θ(???)
- Deletion of max element:
  - Scan the array to find max item.
  - Delete that item.
  - Time: Θ(???)

Naïve Implementation Using Arrays

- Initialization:
  - Given N data, just store them on an array.
  - Time: Θ(N), good!
- Insertion of a new item:
  - (Assumption: the array has enough memory.)
  - Store the item at the end of the array.
  - Time: Θ(1), good!
- Deletion of max element:
  - Scan the array to find max item.
  - Delete that item.
  - Time: Θ(N), bad!

Naïve Implementation Using Lists

- Initialization:
  - Given N data, just store them on an list.
  - Time: Θ(N), good!
- Insertion of a new item:
  - Store the item at the beginning (or end) of the list.
  - Time: Θ(1), good!
- Deletion of max element:
  - Scan the list to find max item.
  - Delete that item.
  - Time: Θ(N), bad!

Using Ordered Arrays/Lists

- Initialization:
  - Given N data, sort them.
  - Time: Θ(???)
- Insertion of a new item:
  - (Assumption: if using an array, it must have enough memory.)
  - Insert the item at the right place, to keep array/list sorted.
  - Time: Θ(???)
- Deletion of max element:
  - Delete the last item.
  - Time: Θ(???)

Using Ordered Arrays/Lists

- Initialization:
  - Given N data, sort them.
  - Time: O(N lg N). OK!
- Insertion of a new item:
  - (Assumption: if using an array, it must have enough memory.)
  - Insert the item at the right place, to keep array/list sorted.
  - Time: O(N). Bad!
- Deletion of max element:
  - Delete the last item.
  - Time: Θ(1). Good!

Using Heaps (New Data Type)

- Initialization:
  - Given N data, **heapify** them (we will see how in a few slides).
  - Time: Θ(N). Good!
- Insertion of a new item:
  - Insert the item at the right place, to maintain the **heap property**. (details in a few slides).
  - Time: O(lg N). Good!
- Deletion of max element:
  - Delete the first item.
  - Rearrange other items, to maintain the **heap property**. (details in a few slides).
  - Time: O(lg N). Good!

Definition of Heaps

- We have two equivalent representations of heaps:
  - As binary trees.
  - As arrays.
- Thus, we have two logically equivalent definitions:
- A binary tree is a heap if, for every node N in that tree, the key of N is larger than or equal to the keys of the children of N, if any.
- An array A (**with 1 as the first index**) is a heap if, for every position N of A:
  - If A[2N] is not out of bounds, then A[N] >= A[2N].
  - If A[2N + 1] is not out of bounds, then A[N] >= A[2*N + 1].

Representing a Heap

- Consider this array:

position	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
value	**X**	**T**	**O**	**G**	**S**	**M**	**N**	**A**	**E**	**R**	**A**	**I**

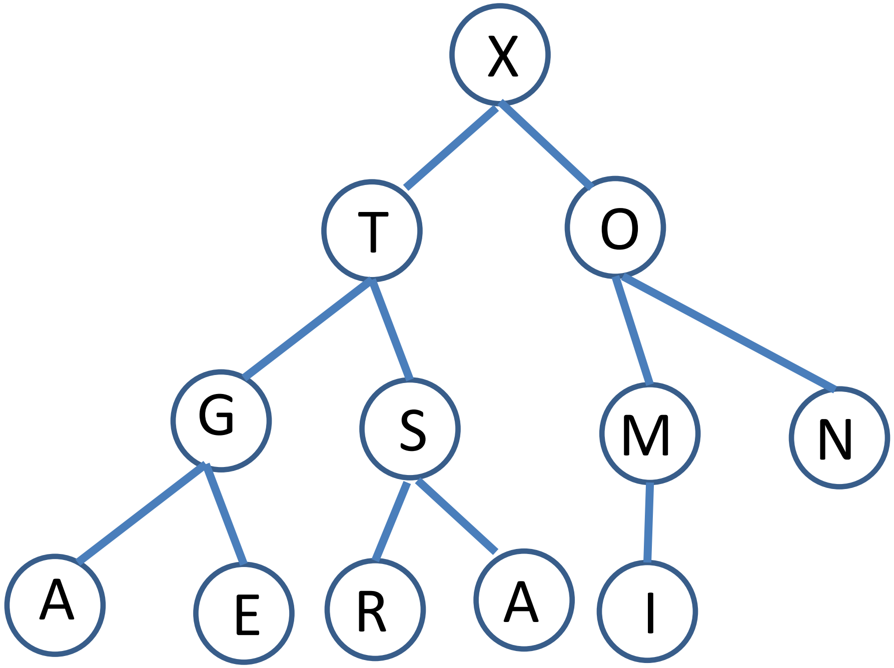
- We can draw the array as a tree.
  - The children of A[N] are A[2N] and A[2N+1].

Representing a Heap

- Consider this array:

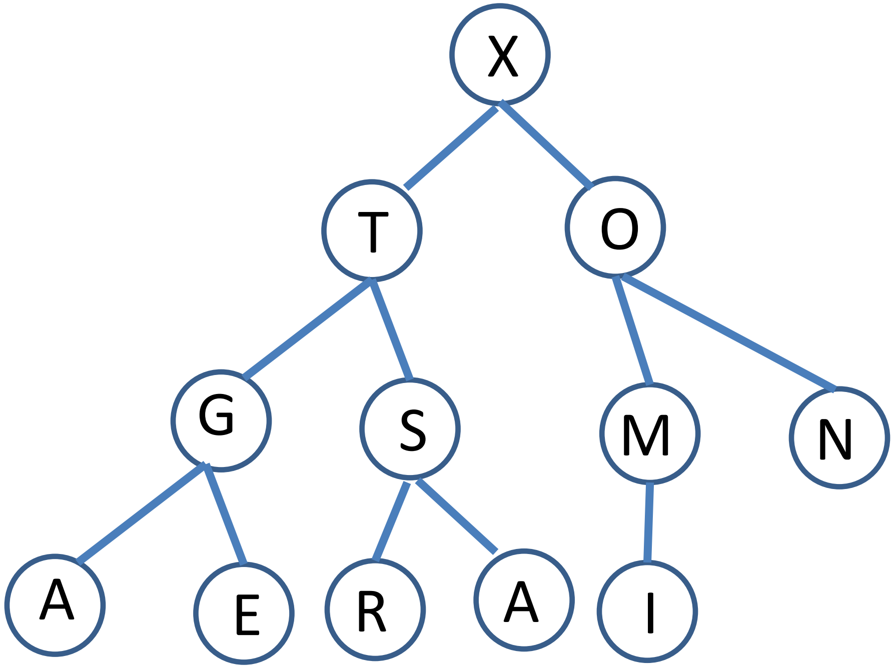
position	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
value	**X**	**T**	**O**	**G**	**S**	**M**	**N**	**A**	**E**	**R**	**A**	**I**

- We can draw the array as a tree.
  - The children of A[N] are A[2N] and A[2N+1].
  - This example shows that the tree and array representations are equivalent.



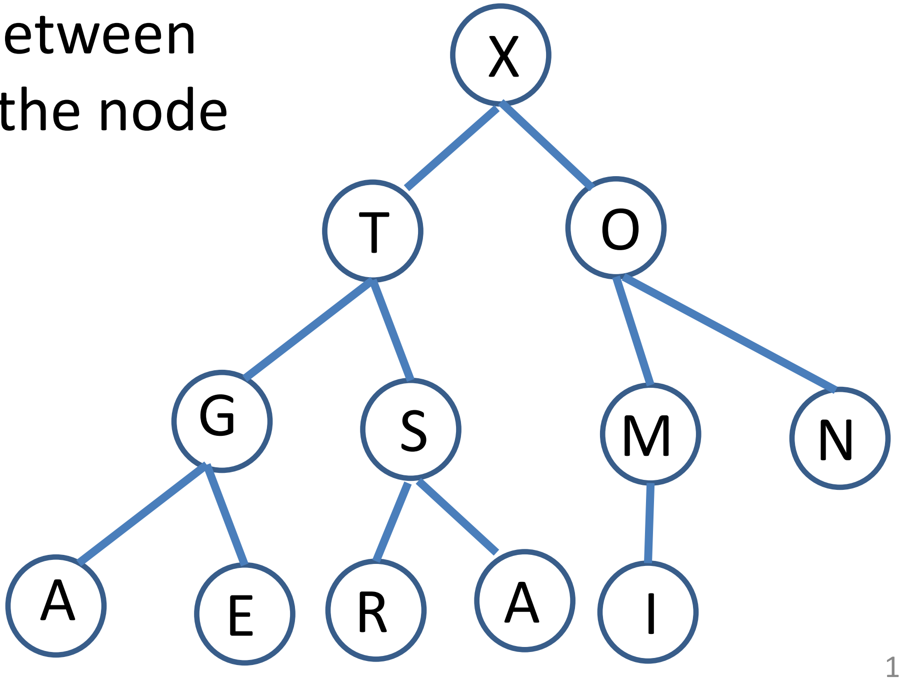
Representing a Heap

- A binary tree representing a heap should be **complete**.
- All levels are full, except possibly for the last level.
- At the last level:
  - Nodes are placed on the left.
  - Empty positions are placed on the right.



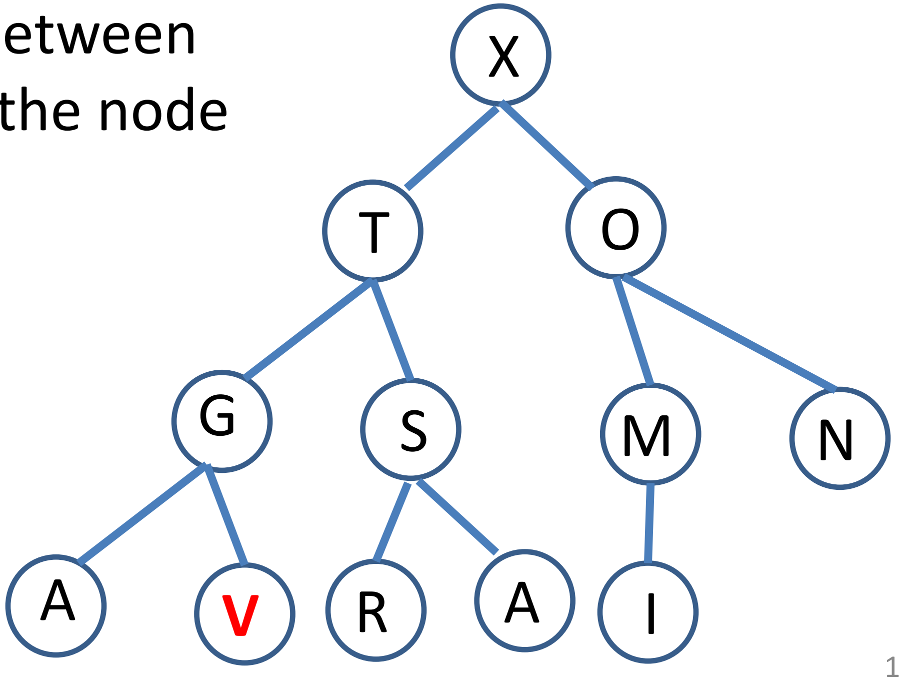
Increasing a Key

- Also called "increasing the priority" of an item.
- Such an operation can lead to violation of the heap property.
- Easy to fix:
  - Exchange items as needed, between node and parent, starting at the node that changed key.



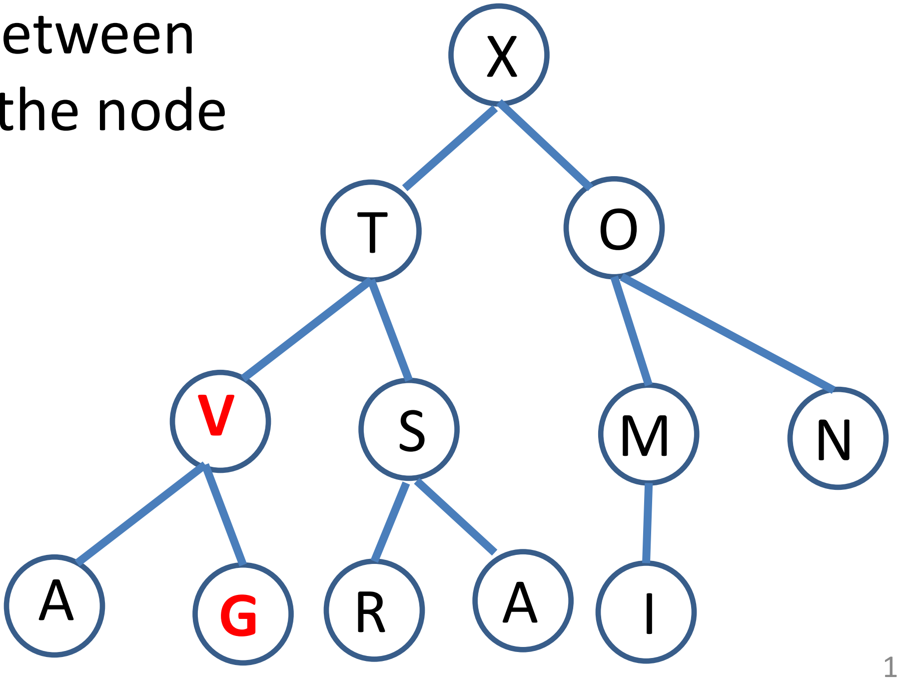
Increasing a Key

- Also called “increasing the priority” of an item.
- Such an operation can lead to violation of the heap property.
- Easy to fix:
  - Exchange items as needed, between node and parent, starting at the node that changed key.
- Example:
  - An E changes to a V.



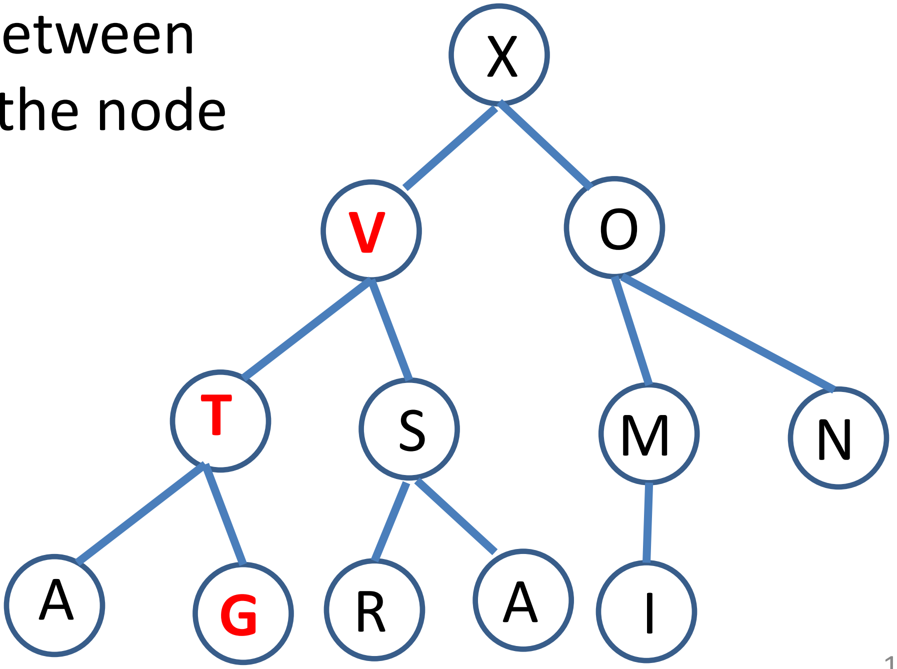
Increasing a Key

- Also called "increasing the priority" of an item.
- Such an operation can lead to violation of the heap property.
- Easy to fix:
  - Exchange items as needed, between node and parent, starting at the node that changed key.
- Example:
  - An E changes to a V.
  - Exchange V and G. Done?



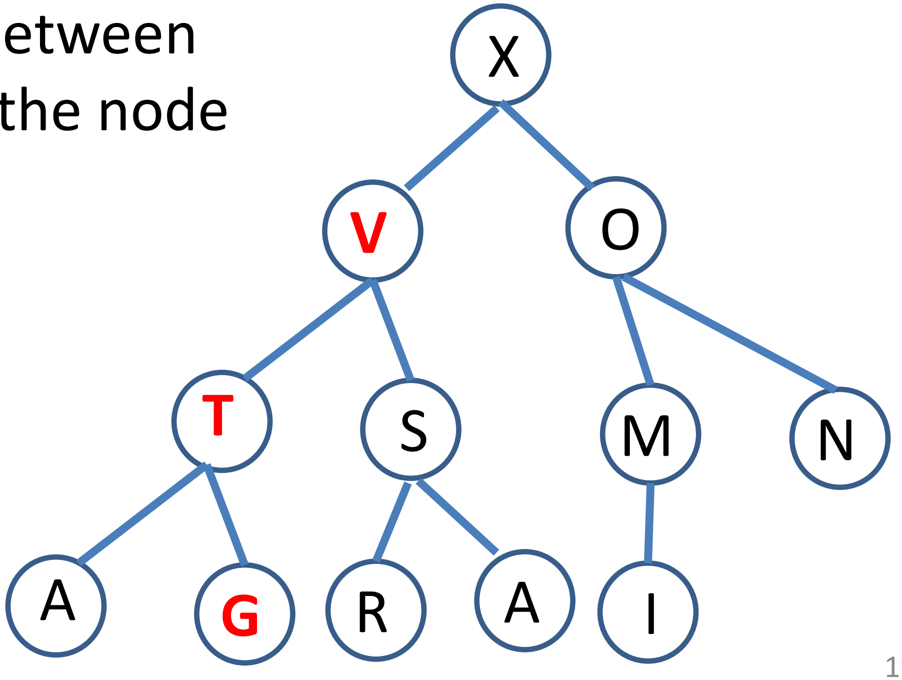
Increasing a Key

- Also called "increasing the priority" of an item.
- Such an operation can lead to violation of the heap property.
- Easy to fix:
  - Exchange items as needed, between node and parent, starting at the node that changed key.
- Example:
  - An E changes to a V.
  - Exchange V and G.
  - Exchange V and T. Done?



Increasing a Key

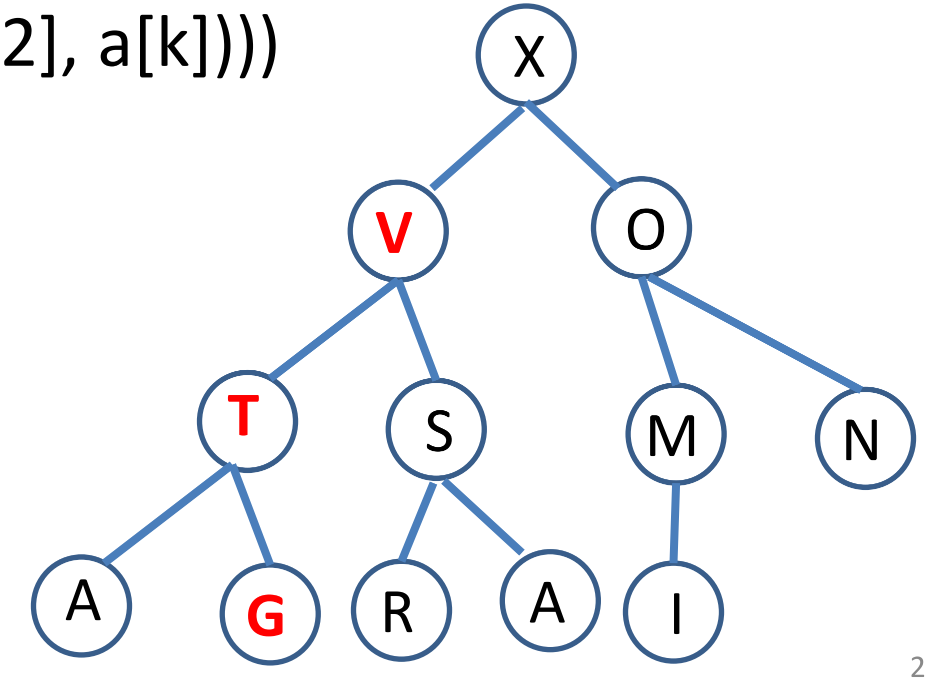
- Also called "increasing the priority" of an item.
- Such an operation can lead to violation of the heap property.
- Easy to fix:
  - Exchange items as needed, between node and parent, starting at the node that changed key.
- Example:
  - An E changes to a V.
  - Exchange V and G.
  - Exchange V and T. Done.



Increasing a Key

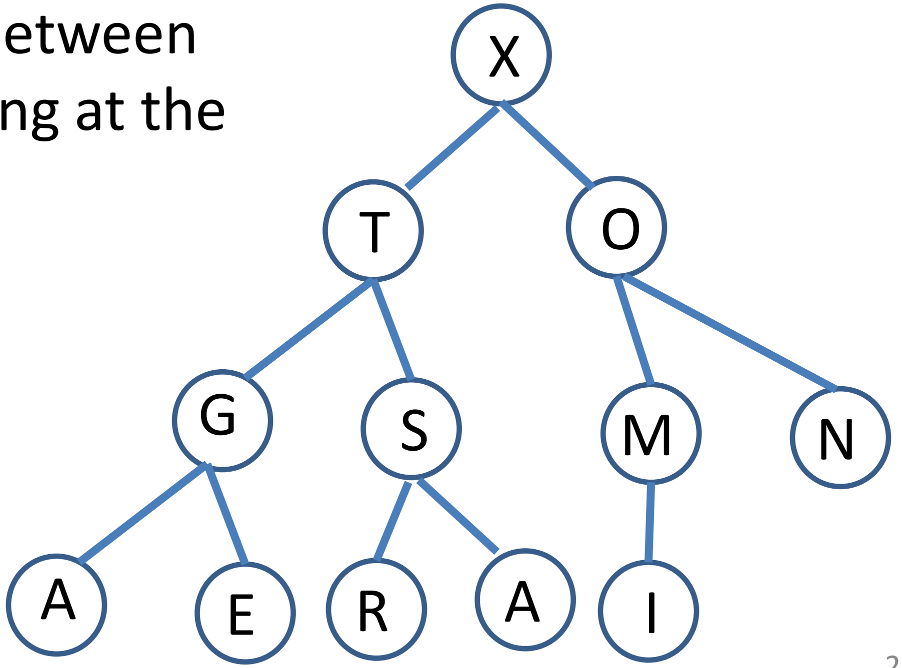
- Implementation:

```
fixUp(Item a[], int k)
{
    while ((k > 1) && (less(a[k/2], a[k])))
    {
        exch(a[k], a[k/2]);
        k = k/2;
    }
}
```



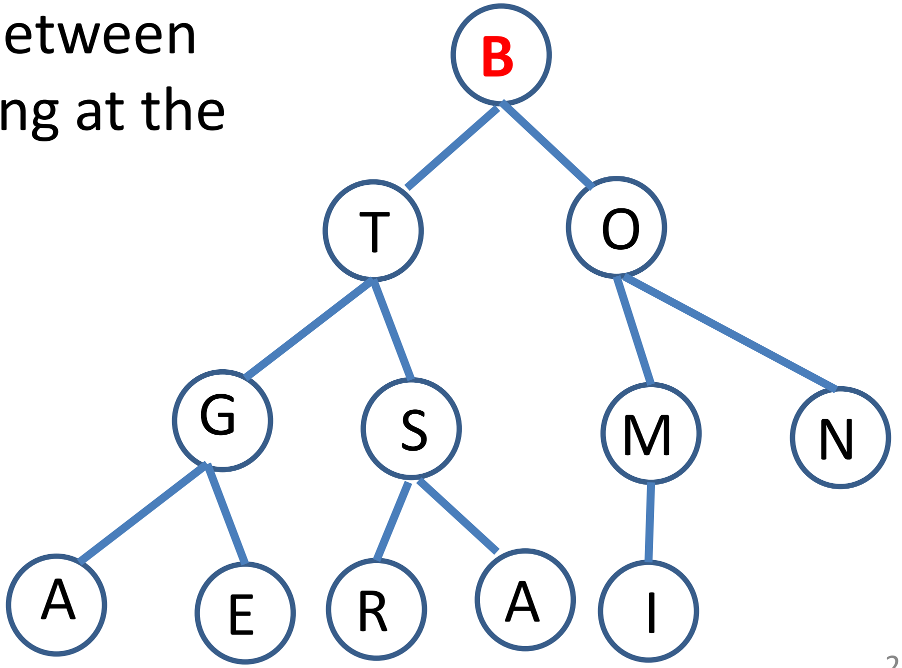
Decreasing a Key

- Also called "decreasing the priority" of an item.
- Such an operation can lead to violation of the heap property.
- Easy to fix:
  - Exchange items as needed, between node and **largest** child, starting at the node that changed key.



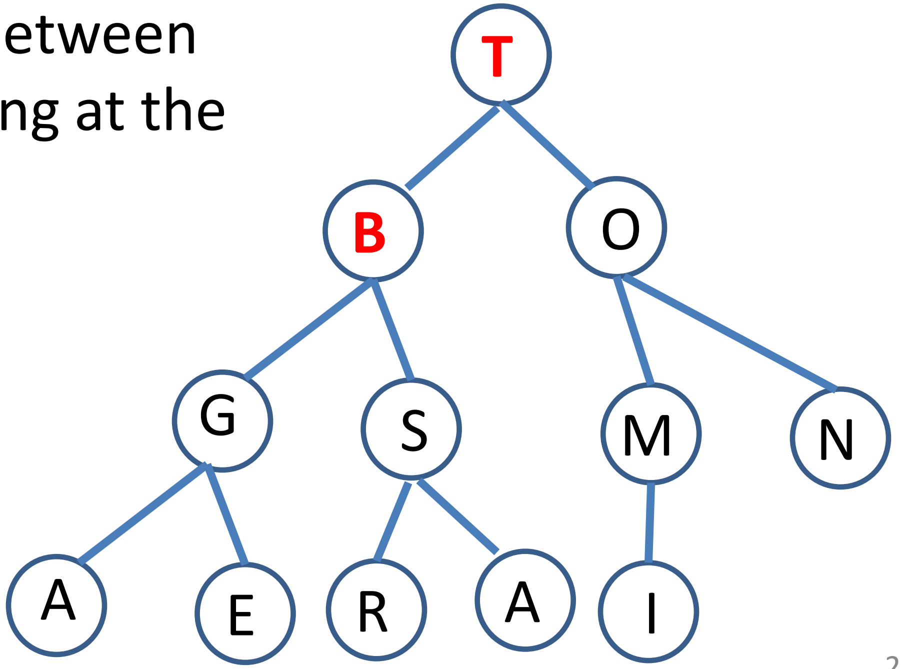
Decreasing a Key

- Also called "decreasing the priority" of an item.
- Such an operation can lead to violation of the heap property.
- Easy to fix:
  - Exchange items as needed, between node and **largest** child, starting at the node that changed key.
- Example:
  - An X changes to a B.



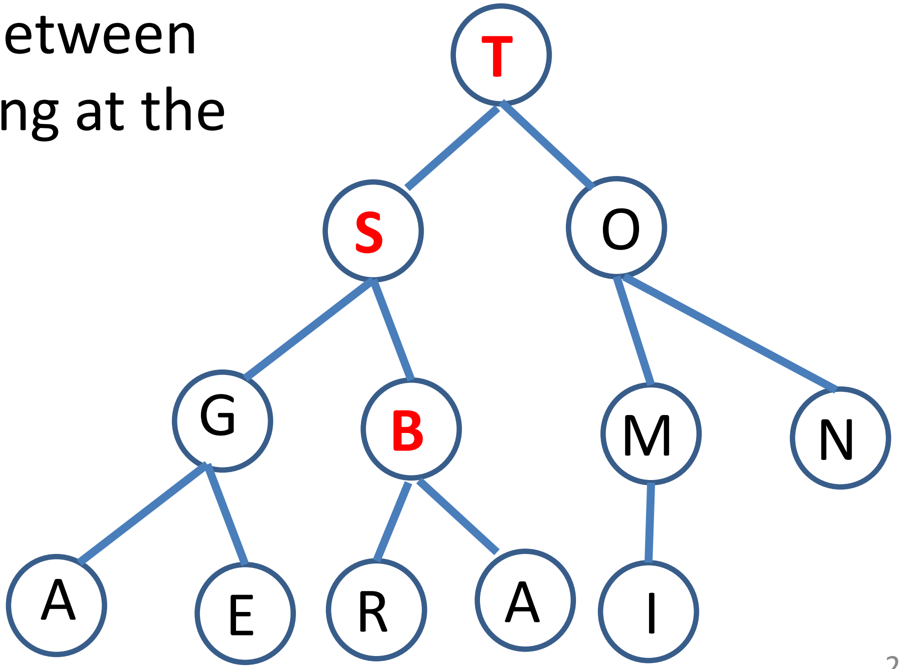
Decreasing a Key

- Also called “decreasing the priority” of an item.
- Such an operation can lead to violation of the heap property.
- Easy to fix:
  - Exchange items as needed, between node and **largest** child, starting at the node that changed key.
- Example:
  - An X changes to a B.
  - Exchange B and T.



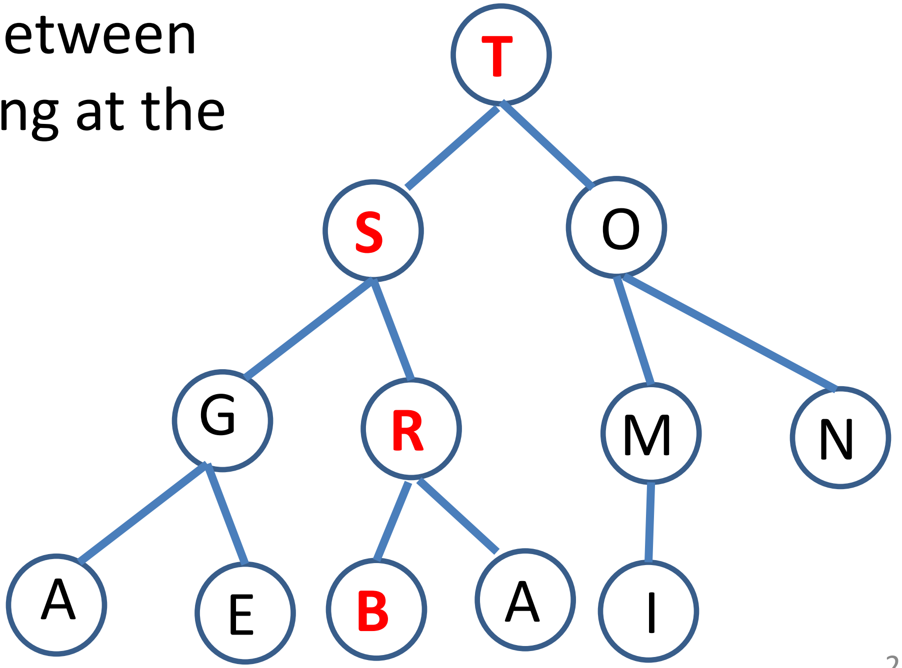
Decreasing a Key

- Also called "decreasing the priority" of an item.
- Such an operation can lead to violation of the heap property.
- Easy to fix:
  - Exchange items as needed, between node and **largest** child, starting at the node that changed key.
- Example:
  - An X changes to a B.
  - Exchange B and T.
  - Exchange B and S.



Decreasing a Key

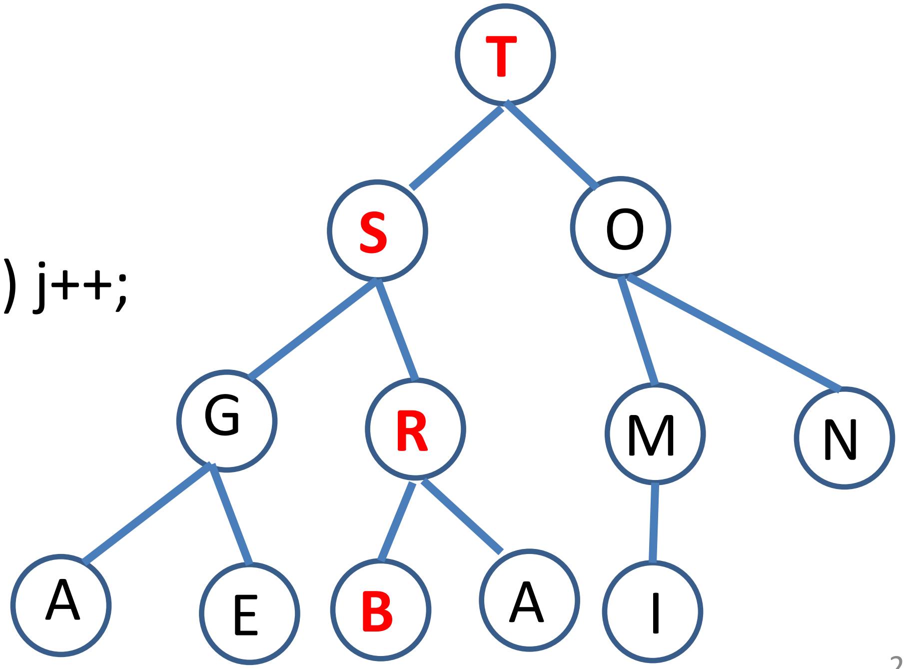
- Also called “decreasing the priority” of an item.
- Such an operation can lead to violation of the heap property.
- Easy to fix:
  - Exchange items as needed, between node and **largest** child, starting at the node that changed key.
- Example:
  - An X changes to a B.
  - Exchange B and T.
  - Exchange B and S.
  - Exchange B and R.



Decreasing a Key

- Implementation:

```
fixDown(Item a[], int k, int N)
{
   int j;
   while (2*k <= N)
   {
      j = 2*k;
      if ((j < N) && less ((a[j], a[j+1]))) j++;
      if (!less(a[k], a[j])) break;
      exch(a[k], a[j]); k = j;
   }
}
```



Insertions and Deletions

- To insert an item to a heap:
  - Insert the item to the end of the heap.
  - Call fix up to restore the heap property.
  - Time = O(???)
- The only element we care to delete from a heap is the maximum element.
- This element is always the first element of the heap.
- To delete the maximum element:
  - Exchange the first and last elements of the heap.
  - Delete the last element (which is the maximum element).
  - Call fixDown to restore the heap property.
  - Time = O(???)

Insertions and Deletions

- To insert an item to a heap:
  - Insert the item to the end of the heap.
  - Call fix up to restore the heap property.
  - Time = O(lg N)
- The only element we care to delete from a heap is the maximum element.
- This element is always the first element of the heap.
- To delete the maximum element:
  - Exchange the first and last elements of the heap.
  - Delete the last element (which is the maximum element).
  - Call fixDown to restore the heap property.
  - Time = O(lg N)

Batch Initialization

- Batch initialization of a heap is the process of converting an unsorted array of data into a heap.
- We will see two methods that are pretty easy to implement:
- **Top-down batch initialization**.
  - O(N lg N) time.
  - O(N) extra space (in addition to the space that the input array already takes).
- **Bottom-up batch initialization**.
  - O(N) time.
  - O(1) extra space (in addition to the space that the input array already takes).

Top-Down Batch Initialization

```
Heap top_down_heap_init(Item * array, int N)
  Heap result = newHeap(N).
  for counter = 0, ..., N-1.
    heap_insert(array[counter]).
  return result.
```

- How much time does this take?

Top-Down Batch Initialization

```
Heap top_down_heap_init(Item * array, int N)
    Heap result = newHeap(N).
    for counter = 0, ..., N-1.
        heap_insert(array[counter]).
    return result.
```

- How much time does this take?
  - We need to do N insertions.
  - Each insertion takes O(lg N) time.
  - So, in total, we need O(N lg N) time.

Bottom-Up Batch Initialization

```
struct heap_struct
{
  int length;
  Item * array;
};
```

```
typedef struct heap_struct * Heap;

Heap bottom_up_heap_init(Item * array, int N)
  for counter = N/2, ..., 1
    fixDown(array, counter, N).

  Heap result = malloc(sizeof(*result)).
  result.array = array.
  result.N = N.
  return result.
```

Visualizing Bottom-Up Initialization

- N = 14
- counter = 7
- fixDown(counter, N):

position	**1**	**2**	**3**	**4**	**5**	**6**	***7**	**8**	**9**	**10**	**11**	**12**	**13**	**<u>14</u>**
value	50	40	30	15	60	10	28	45	35	55	95	90	85	60

Visualizing Bottom-Up Initialization

- N = 14
- counter = 7
- fixDown(counter, N):

position	**1**	**2**	**3**	**4**	**5**	**6**	***7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
value	50	40	30	15	60	10	60	45	35	55	95	90	85	28

Visualizing Bottom-Up Initialization

- N = 14
- counter = 6
- fixDown(counter, N):

position	**1**	**2**	**3**	**4**	**5**	***6**	**7**	**8**	**9**	**10**	**11**	**<u>12</u>**	**<u>13</u>**	**14**
value	50	40	30	15	60	10	60	45	35	55	95	90	85	28

Visualizing Bottom-Up Initialization

- N = 14
- counter = 6
- fixDown(counter, N):

position	**1**	**2**	**3**	**4**	**5**	***6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
value	50	40	30	15	60	90	60	45	35	55	95	10	85	28

Visualizing Bottom-Up Initialization

- N = 14
- counter = 5
- fixDown(counter, N):

position	**1**	**2**	**3**	**4**	***5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
value	50	40	30	15	60	90	60	45	35	55	95	10	85	28

Visualizing Bottom-Up Initialization

- N = 14
- counter = 5
- fixDown(counter, N):

position	**1**	**2**	**3**	**4**	***5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
value	50	40	30	15	95	90	60	45	35	55	60	10	85	28

Visualizing Bottom-Up Initialization

- N = 14
- counter = 4
- fixDown(counter, N):

position	**1**	**2**	**3**	***4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
value	50	40	30	15	95	90	60	45	35	55	60	10	85	28

Visualizing Bottom-Up Initialization

- N = 14
- counter = 4
- fixDown(counter, N):

position	**1**	**2**	**3**	***4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
value	50	40	30	45	95	90	60	15	35	55	60	10	85	28

Visualizing Bottom-Up Initialization

- N = 14
- counter = 3
- fixDown(counter, N):

position	**1**	**2**	***3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
value	50	40	30	45	95	90	60	15	35	55	60	10	85	28

Visualizing Bottom-Up Initialization

- N = 14
- counter = 3
- fixDown(counter, N):

position	**1**	**2**	***3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
value	50	40	90	45	95	30	60	15	35	55	60	10	85	28

Visualizing Bottom-Up Initialization

- N = 14
- counter = 3
- fixDown(counter, N):

position	**1**	**2**	***3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
value	50	40	90	45	95	85	60	15	35	55	60	10	30	28

Visualizing Bottom-Up Initialization

- N = 14
- counter = 2
- fixDown(counter, N):

position	**1**	***2**	**3**	**<u>4</u>**	**<u>5</u>**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
value	50	40	90	45	95	85	60	15	35	55	60	10	30	28

Visualizing Bottom-Up Initialization

- N = 14
- counter = 2
- fixDown(counter, N):

position	**1**	***2**	**3**	**<u>4</u>**	**<u>5</u>**	**6**	**7**	**8**	**9**	**<u>10</u>**	**<u>11</u>**	**12**	**13**	**14**
value	50	95	90	45	40	85	60	15	35	55	60	10	30	28

Visualizing Bottom-Up Initialization

- N = 14
- counter = 2
- fixDown(counter, N):

position	**1**	***2**	**3**	**<u>4</u>**	**<u>5</u>**	**6**	**7**	**8**	**9**	**<u>10</u>**	**<u>11</u>**	**12**	**13**	**14**
value	50	95	90	45	60	85	60	15	35	55	40	10	30	28

Visualizing Bottom-Up Initialization

- N = 14
- counter = 1
- fixDown(counter, N):

position	***1**	**<u>2</u>**	**<u>3</u>**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
value	50	95	90	45	60	85	60	15	35	55	40	10	30	28

Visualizing Bottom-Up Initialization

- N = 14
- counter = 1
- fixDown(counter, N):

position	***1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
value	95	50	90	45	60	85	60	15	35	55	40	10	30	28

Visualizing Bottom-Up Initialization

- N = 14
- counter = 1
- fixDown(counter, N):

position	***1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
value	95	60	90	45	50	85	60	15	35	55	40	10	30	28

Visualizing Bottom-Up Initialization

- N = 14
- counter = 1
- fixDown(counter, N):

position	***1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
value	95	60	90	45	55	85	60	15	35	50	40	10	30	28

Visualizing Bottom-Up Initialization

- N = 14
- counter = 1
- DONE!!!
- The heap condition is now satisfied.

position	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
value	95	60	90	45	55	85	60	15	35	50	40	10	30	28

Running Time

- How can we analyze the running time?
- To simplify, suppose that $N = 2^n - 1$.
- The counter starts at value ???.
- At that point, we call fixDown on a heap of size ???.

```
Heap bottom_up_heap_init(Item * array, int N)
  for counter = N/2, ..., 1
    fixDown(array, counter, N).

  Heap result = malloc(sizeof(*result)).
  result.array = array.
  result.N = N.
  return result.
```

Running Time

- How can we analyze the running time?
- To simplify, suppose that $N = 2^n - 1$.
- The counter starts at value $2^{n-1} - 1$.
- At that point, we call fixDown on a heap of size 3 (= $2^2 - 1$).
- For counter values between $2^{n-1} - 1$ and 2^{n-2}, we call fixDown on a heap of size $2^2 - 1$.

```
Heap bottom_up_heap_init(Item * array, int N)
  for counter = N/2, ..., 1
    fixDown(array, counter, N).

  Heap result = malloc(sizeof(*result)).
  result.array = array.
  result.N = N.
  return result.
```

Running Time

- For counter values between 2^{n-1} - 1 and 2^{n-2}, we call fixDown on a heap of size 2^2 - 1.
- For counter values between 2^{n-2} - 1 and 2^{n-3}, we call fixDown on a heap of size ???.
- …
- For counter value 2^0 we call fixDown on a heap of size ???.

```
Heap bottom_up_heap_init(Item * array, int N)
  for counter = N/2, ..., 1
    fixDown(array, counter, N).

  Heap result = malloc(sizeof(*result)).
  result.array = array.
  result.N = N.
  return result.
```

Running Time

- For counter values between $2^{n-1} - 1$ and 2^{n-2}, we call fixDown on a heap of size $2^2 - 1$.
- For counter values between $2^{n-2} - 1$ and 2^{n-3}, we call fixDown on a heap of size 7 (= $2^3 - 1$).
- …
- For counter value 2^0 we call fixDown on a heap of size $2^n - 1$.

```
Heap bottom_up_heap_init(Item * array, int N)
  for counter = N/2, ..., 1
    fixDown(array, counter, N).

  Heap result = malloc(sizeof(*result)).
  result.array = array.
  result.N = N.
  return result.
```

Running Time

Counter: from	Counter: to	Number of Iterations	Heap Size	Time per Iteration	Time for All Iterations
$2^{n-1} - 1$	2^{n-2}	2^{n-2}	$2^2 - 1$	O(2)	$O(2^{n-2} * 2)$
$2^{n-2} - 1$	2^{n-3}	2^{n-3}	$2^3 - 1$	O(3)	$O(2^{n-3} * 3)$
$2^{n-3} - 1$	2^{n-4}	2^{n-4}	$2^4 - 1$	O(4)	$O(2^{n-4} * 4)$
…					
$2^1 - 1=1$	$2^0 = 1$	$2^0 = 1$	$2^n - 1$	O(n)	$O(2^0 * n)$

- Sum: $\sum_{k=0}^{n-2}(2^k * (n-k))$
- This is not that trivial to analyze.
- It turns out that $\sum_{k=0}^{n-2}(2^k * (n-k))$ = Θ(N)
- Thus, bottom-up batch initialization takes linear time.

Bottom-Up Versus Top-Down

- Top-down initialization does not touch the input array.
  - Instead, it creates a new heap, where it inserts the data.
  - Thus, it needs O(N) extra space, in addition to the space already taken by the input array.
- Bottom-up initialization, instead, changes the input array.
  - The heap does not allocate memory for a new array.
  - Instead, the heap uses the input array as its own array.
  - Consequently, it needs O(1) extra space, in addition to the space already taken by the input array.

Heapsort

```
void heapsort(Item a[], int N)
    bottom_up_heap_init(a, N).
    for counter = N, …, 2
        exch(a[1], a[counter]).
        fixDown(a, 1, counter-1).
```