Event Calendars (Lists) $\mathcal{E}$

October 30, 2025

- Large NES may have *thousands* of events in the event list
- Such a NES will spend most of CPU time on managing event list
- Efficient event management will reduce overall CPU time
- Structures also applicable to SJF queue discipline

Introduction

- Event list: data structure of events, plus any extra associated info
- Elements in the list: *event notices*
- List size classifications:
  - Fixed maximum. No need for dynamic memory allocation.
  - Variable or unknown maximum.
- Application:
  - A specific model. We can exploit model characteristics.
  - General-purpose (e.g., simulation language). Need robust DS

Event list operations

- Critical operations:
  - Insert / enqueue
    - "Schedule" an event
  - Delete / dequeue
    - Usually: process the next event
    - Rarely: cancel an already-scheduled event
- Other operations (not considered here):
  - Change operation
    - Search to change an attribute (e.g., scheduled time)
  - Examine operation
    - Search to read an attribute
  - Count operation
    - How many event notices in list?

Event list criteria

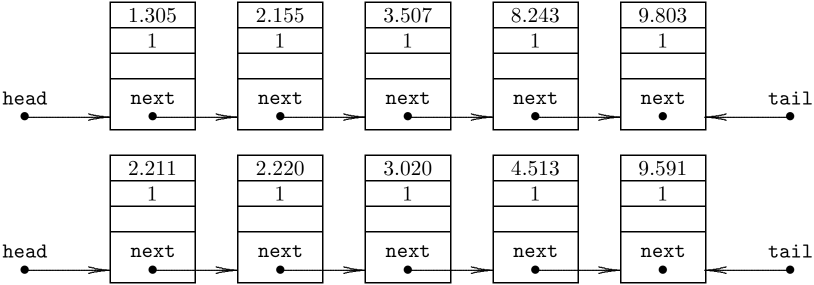
- Speed
  - Balance between sophisticated DS and overhead
  - Consider average-case and worst-case performance
- Robustness
  - Performs well for many scheduling scenarios
  - General purpose: performs well for many simulation models
    - Use diverse, representative benchmark models
- Adaptability
  - Adapt to changes in event distributions
  - Adapt to changes in event list size
  - Parameter-free

Advanced event list management

- Further discussion is for general case:
  1. Number of events in list varies
  2. Maximum size of event list is unknown
  3. Structure of simulation model is unknown
- We will discuss four structures:
  1. Multiple linked lists
  2. Binary search trees
  3. Heaps
  4. Hybrid schemes
- Other structures exist

Multiple Linked Lists

- Use k ordered lists
- Insert into shortest list: worst case $\mathcal{O}(n/k)$
- Delete is $\mathcal{O}(k)$: check front of each list
- Example for $k = 2$, $n = 10$ for `ttr`:



Issues for Multiple Linked Lists

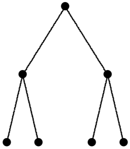
- Should k be fixed, or allowed to vary?
- If fixed, what is a good k?
- If variable, decide
  - When to increase or decrease k (as a function of n)
  - How to increase k: split lists or start new?
  - How to decrease k: merge lists?

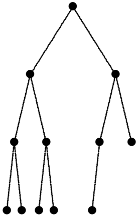
Binary trees

Recall:

- Each node has at most 2 child nodes, and at most one parent node
- The node with no parent: *root*
- A node with no children: *leaf*
- Node *level*: 1 if root, 1+parent's level otherwise
- Tree *height*: maximum level

More on binary trees



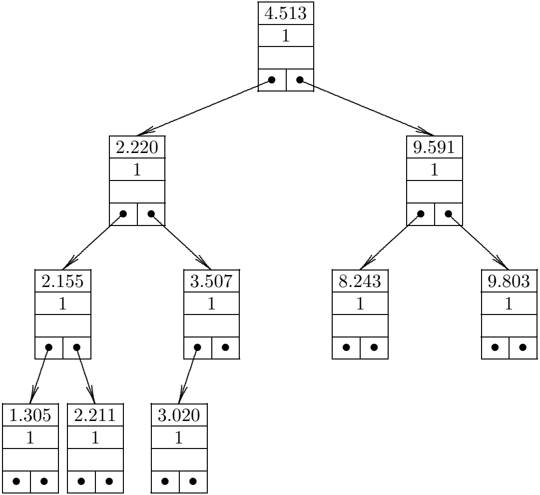


Full tree:

- All leaves at same level
- All non-leaves have 2 children

Complete tree:

- Full down to level $h - 1$
- Level h is filled “left to right”



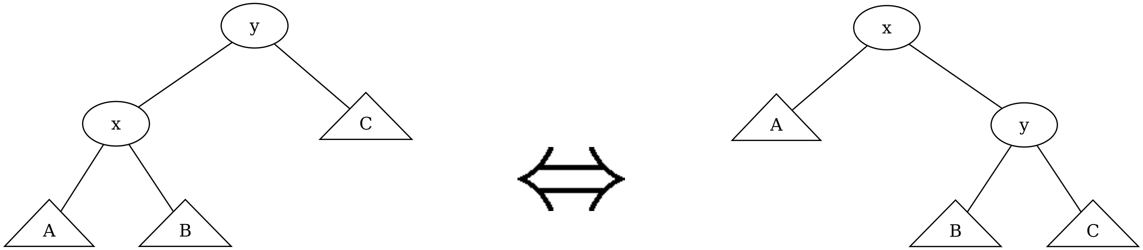
- Node property: Left child value $\leq$ Node value $\leq$ Right child value

Binary search trees for event lists

- Leftmost node is most imminent
- Worst-case insert: $\mathcal{O}(h)$
- Worst-case delete: $\mathcal{O}(h)$
- Unbalanced trees: easier to implement
  - Height of tree h can be as large as n
- Balanced trees: need to rotate nodes
  - Height can be limited to $h = \mathcal{O}(\log n)$
  - AVL, red-black, Splay trees are examples

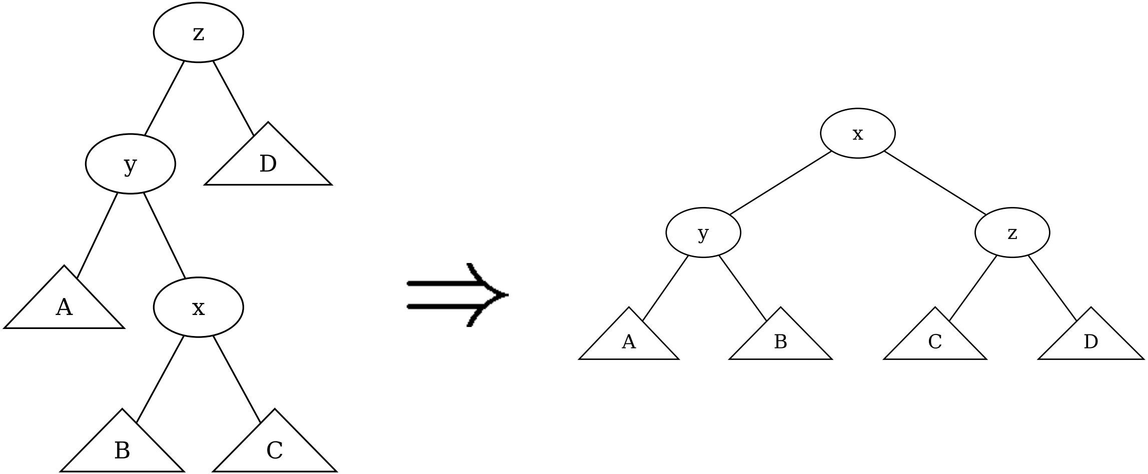
Splay Tree Rotations (A, B, C & D are Subtrees)

"Standard Rotation" Moves *x* to its Parent's Location



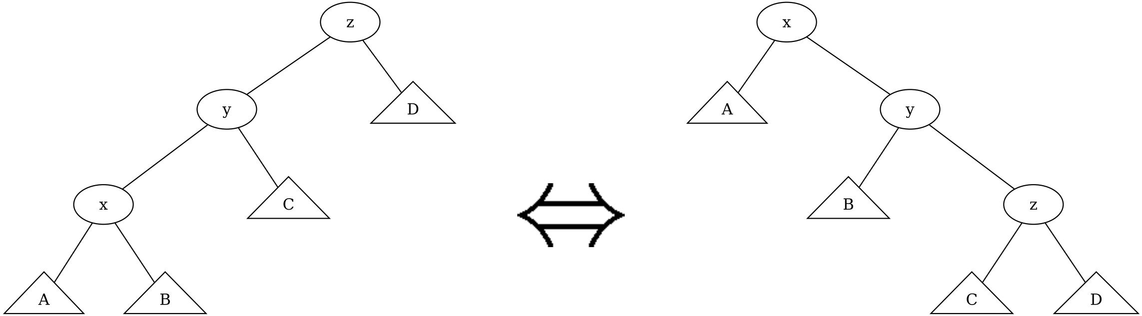
Splay Tree Rotations (A, B, C & D are Subtrees)

"Zig-Zag" Moves *x* to its Grandparent's Location



Splay Tree Rotations (A, B, C & D are Subtrees)

"Zig-Zig" Moves *x* to its Grandparent's Location



Splay Tree Operations

Philosophy: by splaying a recently accessed node to the root, keep "nearby nodes" near the top of the tree. Any node can be splayed to the root with a combination of to-grandparent Zig operations and perhaps one standard rotation.

Insertion Standard BST insertion, splay the inserted node to the root.

Deletion Standard BST deletion (recall non-trivial deletion: replace to-be-removed node with "leftmost of right tree" or "rightmost of left tree"), splay the *parent* of deleted node to the root.

Search Splay the found node to the root.

- Amortized equivilant performance ($\log n$) **with no additional memory overhead**.
- You can **always** splay simple BSTs

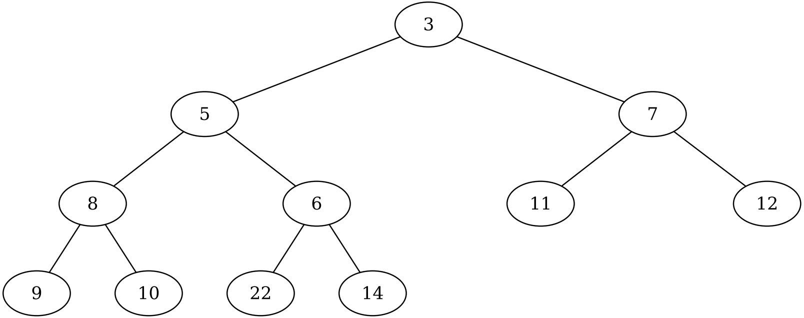
Heaps as event lists

- Always a *complete* binary tree
- Root node is most imminent
- Insert, delete: maintain heap property by swapping nodes with parent
  - Easier to implement than balanced binary search trees
- Worst-case insert: $\mathcal{O}(\log n)$
- Worst-case delete: $\mathcal{O}(\log n)$
- Searching for arbitrary event: $\mathcal{O}(n)$

Priority Queues ("Heaps")

index (i)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
value	3	5	7	8	6	11	12	9	10	22	14				

$$\underbrace{leftChild(i) = 2i+1 \quad rightChild(i) = 2i+2}_{i \geq 0} \quad parent(i) = \left\lfloor \frac{i-1}{2} \right\rfloor \quad i > 0$$

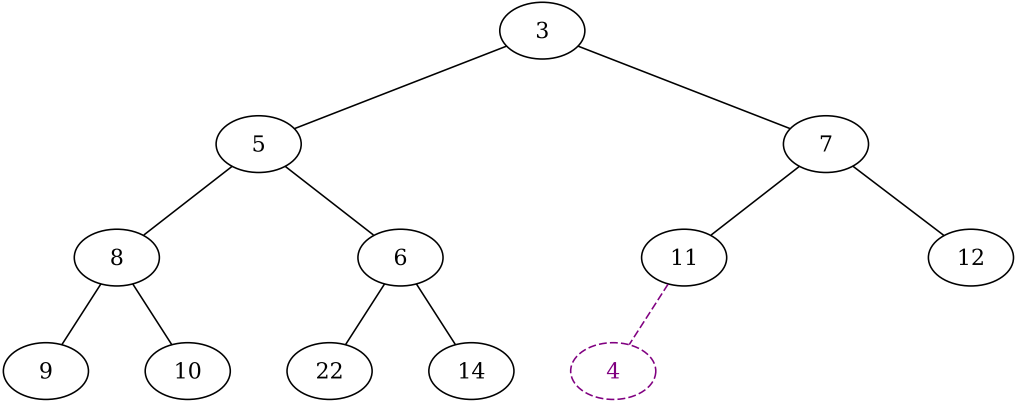


Invariant Property: Any parent has higher priority (**smaller activation time**) than either of its children (ie: all descendents).

Heap definition: using specific access equations on an array (vector) so that it can be thought to hold a complete binary tree in memory (given n elements).

Priority Queues — Insert 4 (*enqueue event*)

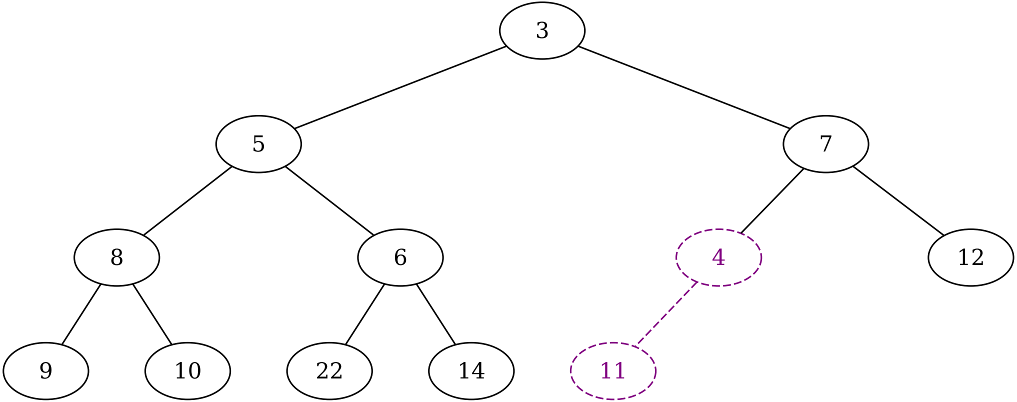
Place new entry in first empty leaf slot, may require increasing heap memory allocation.



Invariant Property: Any parent has higher priority (**smaller activation time**) than either of its children (ie: all descendents).

Priority Queues — Insert 4 (*enqueue event*)

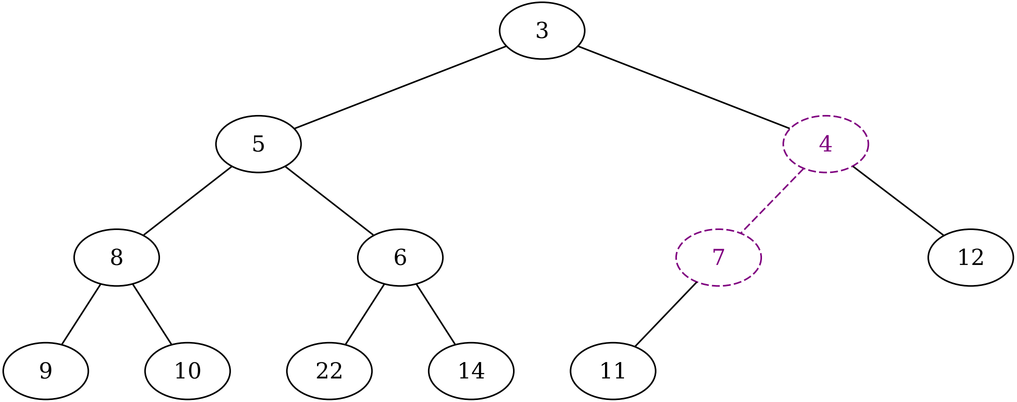
"Bubble," "percolate," or push up: swap with node 11 to maintain invariant property.



Invariant Property: Any parent has higher priority (**smaller activation time**) than either of its children (ie: all descendents).

Priority Queues — Insert 4 (*enqueue event*)

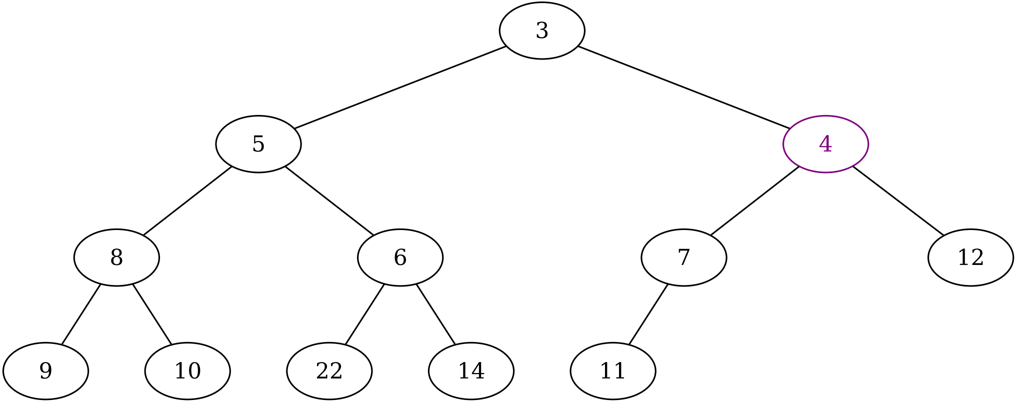
"Bubble," "percolate," or push up: swap with node 7.



Invariant Property: Any parent has higher priority (**smaller activation time**) than either of its children (ie: all descendents).

Priority Queues — Insert 4 (*enqueue event*)

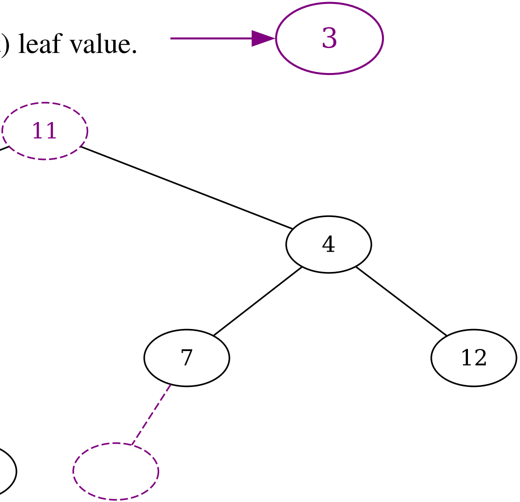
Finished, priority queue invariant property restored.



Invariant Property: Any parent has higher priority (**smaller activation time**) than either of its children (ie: all descendents).

Priority Queues — pop (*dequeue* imminent event)

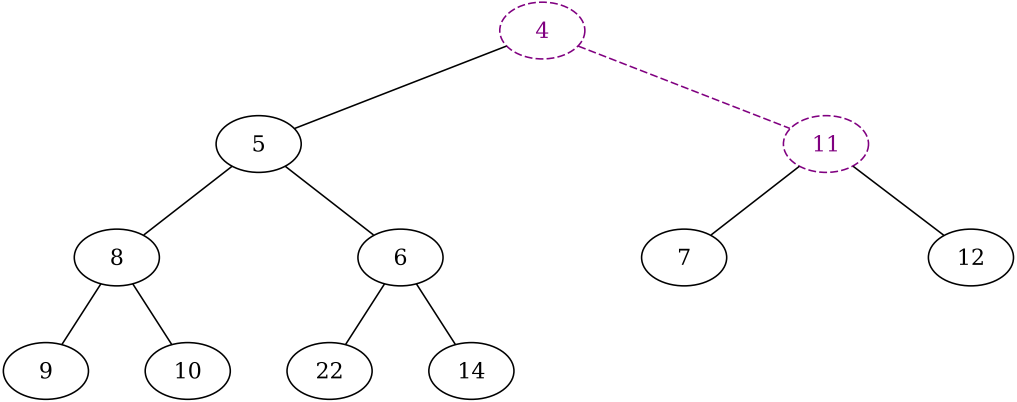
Remove root, replace with "right-most" (last) leaf value.



Invariant Property: Any parent has higher priority (**smaller activation time**) than either of its children (ie: all descendents).

Priority Queues — pop (*dequeue* imminent event)

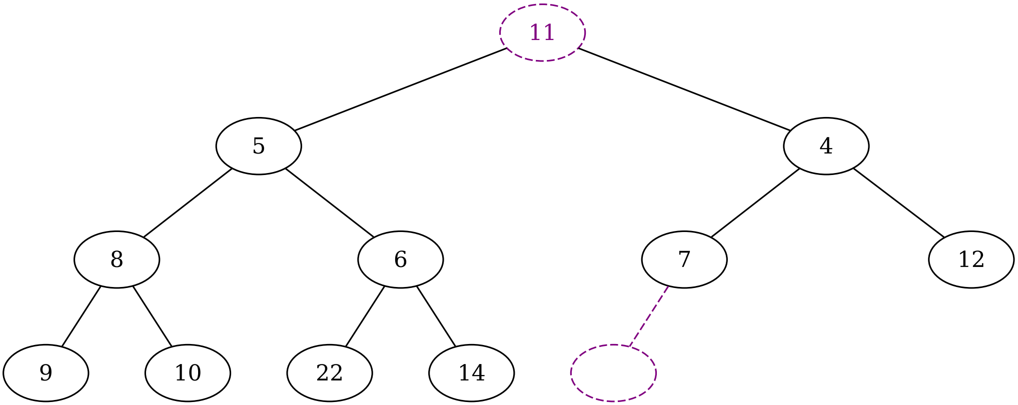
Push down: swap with node 4 to maintain invariant property (always swap with the higher priority child).



Invariant Property: Any parent has higher priority (**smaller activation time**) than either of its children (ie: all descendents).

Priority Queues — pop (*dequeue* imminent event)

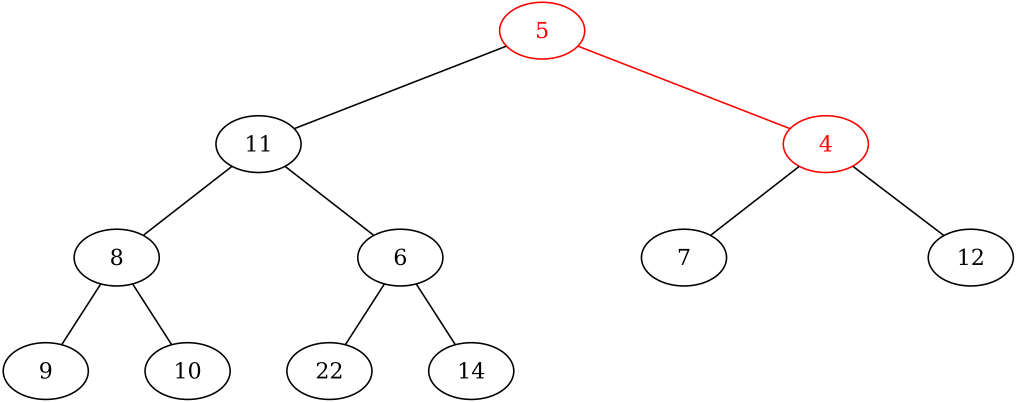
Why not swap with 5 and then 6?



Invariant Property: Any parent has higher priority (**smaller activation time**) than either of its children (ie: all descendents).

Priority Queues — pop (*dequeue* imminent event)

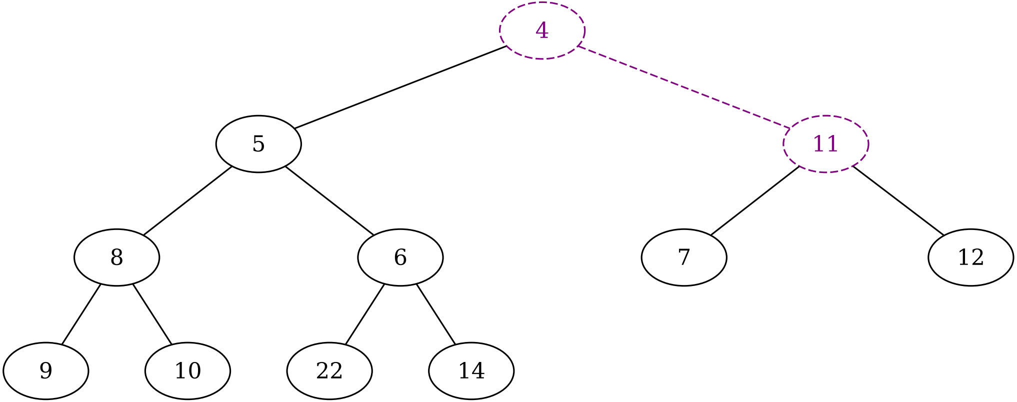
Why not swap with 5 and then 6?



Invariant Property: Any parent has higher priority (**smaller activation time**) than either of its children (ie: all descendents).

Priority Queues — pop (*dequeue* imminent event)

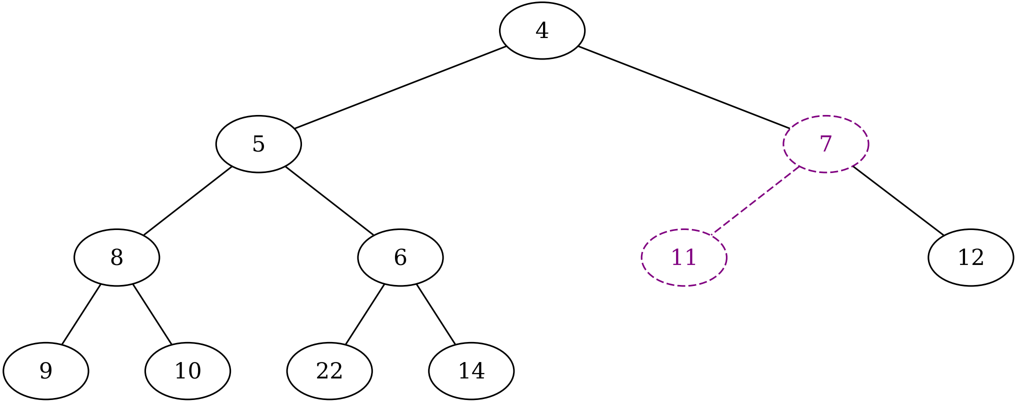
Push down: swap with node 4 to maintain invariant property (always swap with the higher priority child).



Invariant Property: Any parent has higher priority (**smaller activation time**) than either of its children (ie: all descendents).

Priority Queues — pop (*dequeue* imminent event)

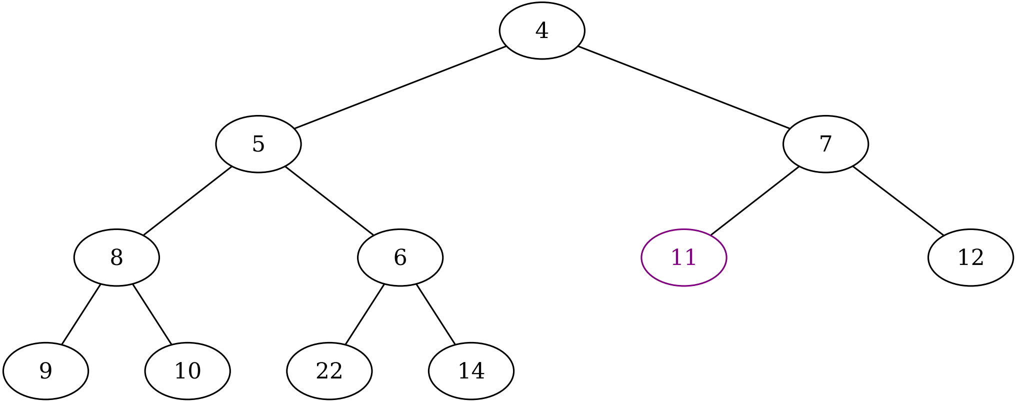
Push down: swap with node 7.



Invariant Property: Any parent has higher priority (**smaller activation time**) than either of its children (ie: all descendents).

Priority Queues — pop (*dequeue* imminent event)

Finished, priority queue invariant property restored.



Invariant Property: Any parent has higher priority (**smaller activation time**) than either of its children (ie: all descendents).

C++ std::priority_queue Event operator<()

```
class Event {
    :
    // C++ requirement: strict weak ordering
    bool operator<( const Event& rhs ) const
    {
        // .at is activation time of the event
        if( this->at == rhs.at) {
            // the comparison of type is arbitrary
            // either < or > will work, but NOT <=, >=
            return this->type < rhs.type;
        }
        // inverted!  we want the least
        // activation time to have
        // higher priority
        return ( this->at > rhs.at );
    }
    :
};
```

Java `java.util.PriorityQueue` Event `compareTo()`

```
public class Event implements Comparable<Event> {
    :
    :
    @Override
    public int compareTo( Event rhs )
    {
        if( this.at == rhs.at ) {
            // if you have simple enumerated types...
            return this.type - rhs.type;
            // or if follow the "everything must be
            // a class" philosophy
            // return this.type.compareTo( rhs.type );
        }
        if( this.at < rhs.at ) return -1;
        return 1;
    }
}
```

Python queue.PriorityQueue Event __lt__()

```
class Event :
    :
    def __lt__( self, rhs ) :
        if self.at == rhs.at :
            return self.type < rhs.type
        return self.at < rhs.at
    :
```

Hybrid schemes

- If n is small, a simple structure may work best
- If n is large, a tree structure should work well
- One hybrid scheme: change data structures. E.g.:
  - When n increases above 15, change to heap
  - When n decreases below 6, change to ordered list

Think-Type-Receive (text section 5.3.3)

A timesharing computer system (as in `isengard.mines.edu`) with N (`ssh`) users connected. Each user behaves in a similar fashion:

i. They **think** for $Uniform(0, 10)$ s
ii. They **type** $Equilikely(5, 15)$ keystrokes, each taking $Uniform(0.15, 35)$ s
iii. They **receive** a response of $Equilikely(50, 300)$ characters each requiring $1/120$ s for transmission (ok, not *quite* `ssh`)

Probability of a user being in a particular state:

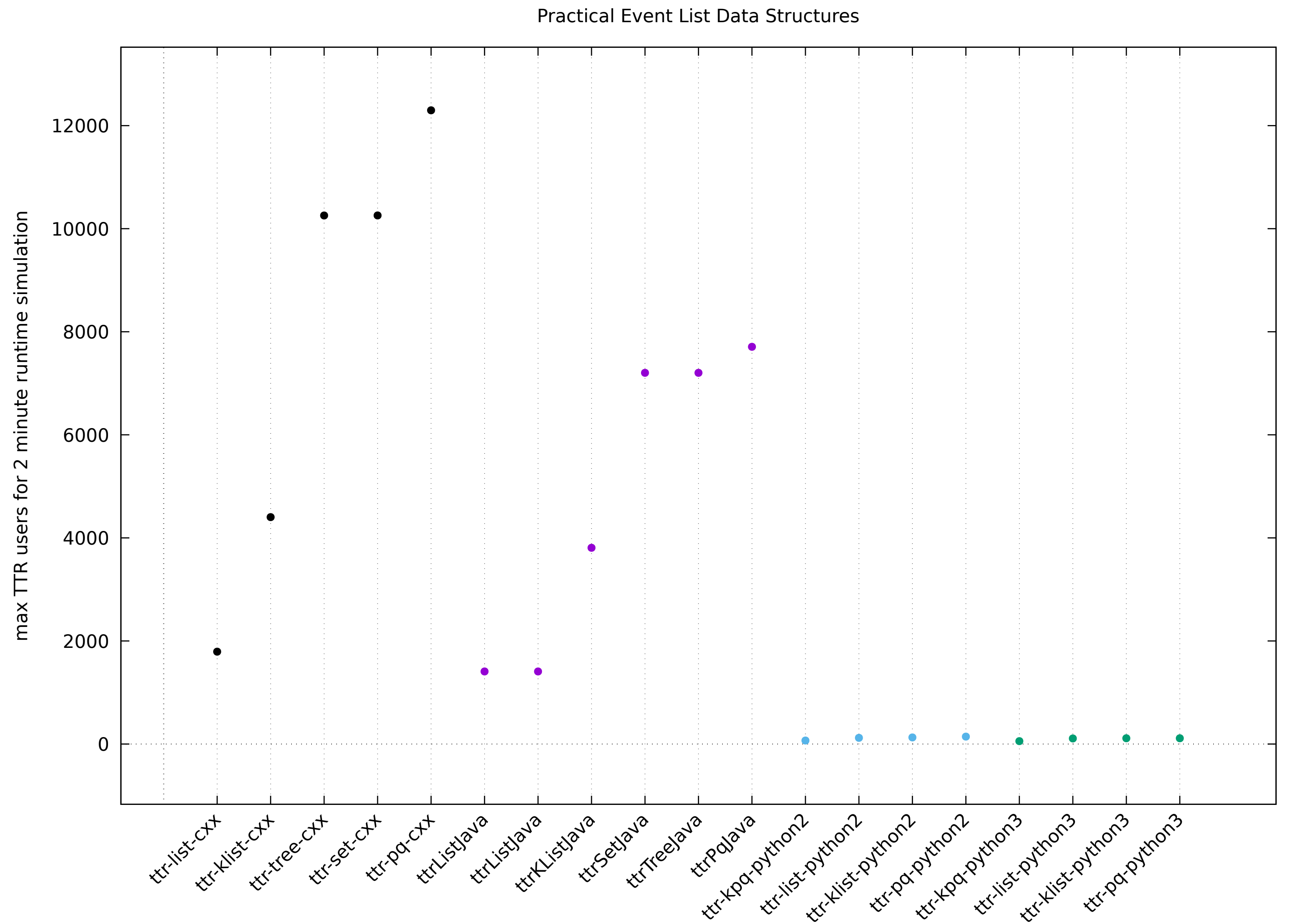
$$\text{thinking} \;\approx 0.56 \quad \text{typing} \;\approx 0.28 \quad \text{receiving} \;\approx 0.16$$

Expected length of a think-type-receive cycle: 8.9583 s

Number of events generated per user (on average, per cycle): 186, or

$$\approx 20 \text{ per simulation second}$$

We can easily envision a similar simulation for modern cloud computing


Practical Event List Data Structures
max TTR users for 2 minute runtime simulation
12000
10000
8000
6000
4000
2000
0
ttr-list-cxx
ttr-klist-cxx
ttr-tree-cxx
ttr-set-cxx
ttr-pq-cxx
ttrListJava
ttrListJava
ttrKListJava
ttrSetJava
ttrTreeJava
ttrPqJava
ttr-kpq-python2
ttr-list-python2
ttr-klist-python2
ttr-pq-python2
ttr-kpq-python3
ttr-list-python3
ttr-klist-python3
ttr-pq-python3

Hendricksen's algorithm

- Uses binary search tree and ordered list *simultaneously*
- Ordered list:
  - Doubly linked
  - Ordered by event time
  - Contains "dummy" events with time $-\infty$ and ∞
- Binary tree:
  - Nodes for a *subset* of event times
  - Node format:

Pointer to next lower time tree node
Pointer to left child tree node
Pointer to right child tree node
Event time
Pointer to the event notice

Operations using Hendricksen's algorithm

- Deletion: from front of list, tree is not "fixed", $\mathcal{O}(1)$
- Insertion (time t):
  1. Find smallest time larger than t in tree
  2. Search backwards in list at most l positions (usually $l = 4$)
  3. Position found: insert
  4. Position not found: "pull" operation to rebalance tree:
     - Go to "next lower time" node in tree
     - Change that tree node to point to current list entry
     - Continue search at step (2)
     - If "next lower time" node not present, add a new level to tree
- Tends to have short average insertion time
- Implemented in simulation languages: GPSS, SLX, SLAM

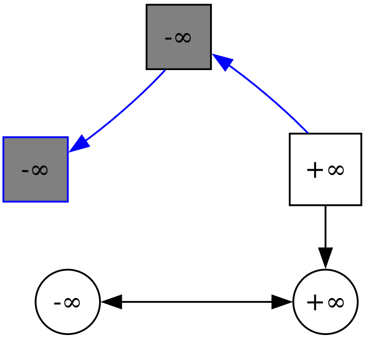
Hendricksen's Algorithm

Tree Node Connections

next smallest $\rightarrow$
pointer into event list $\rightarrow$
copy of `.at` from $\rightarrow$

a. technically, $\rightarrow$ is not required in node, can be derived from heap index.
b. `.at` and $\rightarrow$ (pointer) not always valid
c. pointers to $-\infty$ not shown to reduce edge clutter in graphics
d. "left child," "right child" edges also not shown, but relationship can be seen by relative position of nodes and $\rightarrow$

Initial Structure

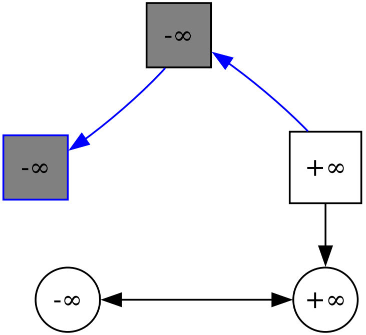


Hendricksen's Algorithm

Tree Node Connections

next smallest $\rightarrow$
pointer into event list $\rightarrow$
copy of `.at` from $\rightarrow$

Initial Structure



e. Tree node copy of **activation time** (`.at`) treated as $-\infty$ if $< t$ (simulation clock time).

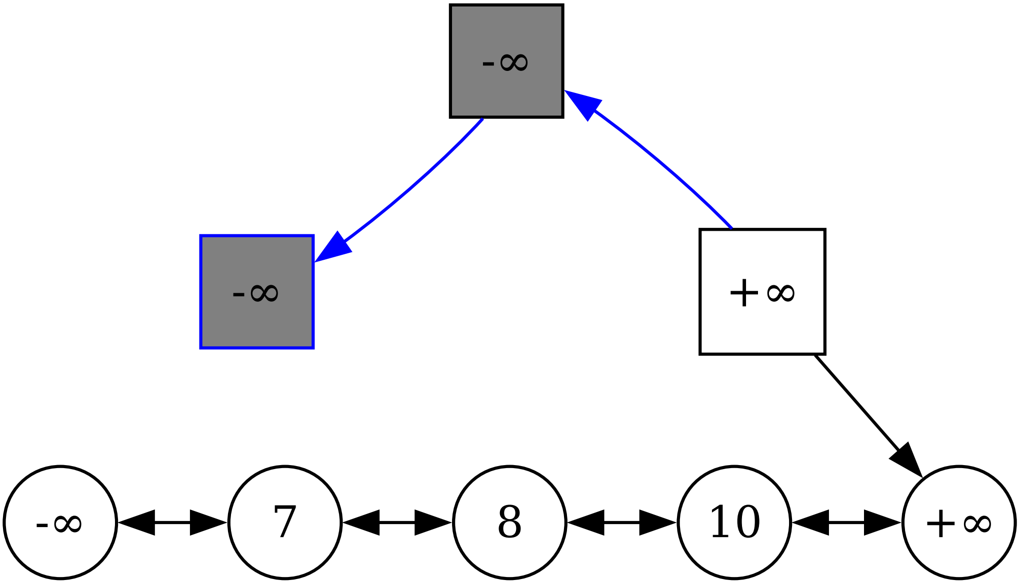
f. Traversal decision unlike standard BST

$$followBranch = \begin{cases} leftChild & \text{if } node.at > t \text{ and } node.at > event.at \\ rightChild & \text{otherwise} \end{cases}$$

IMPORTANT! Tree traversal always seeks (and remembers during traversal) the smallest *node.at* **larger than** *event.at*. If no such node is found, begin list search at $+\infty$.

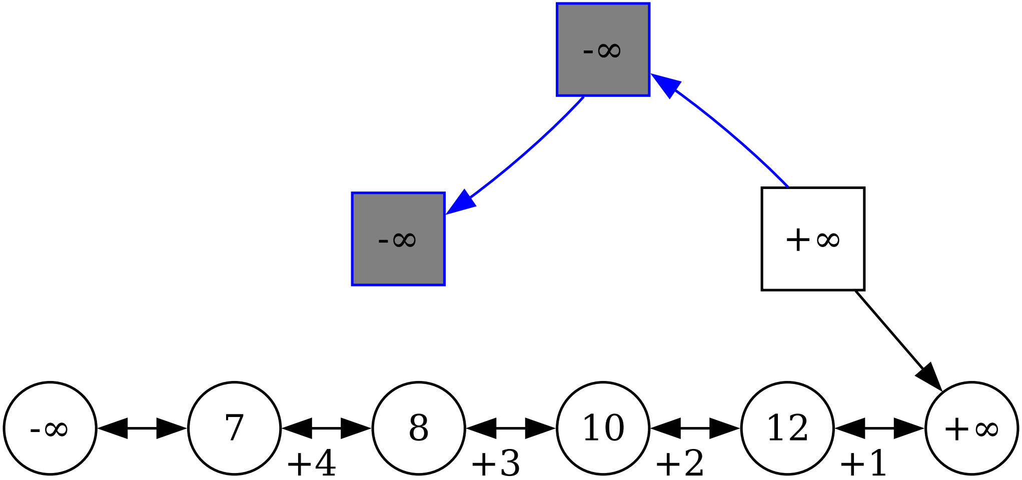
Henriksen's Algorithm

After Inserting 8, 10, then 7



Henriksen's Algorithm

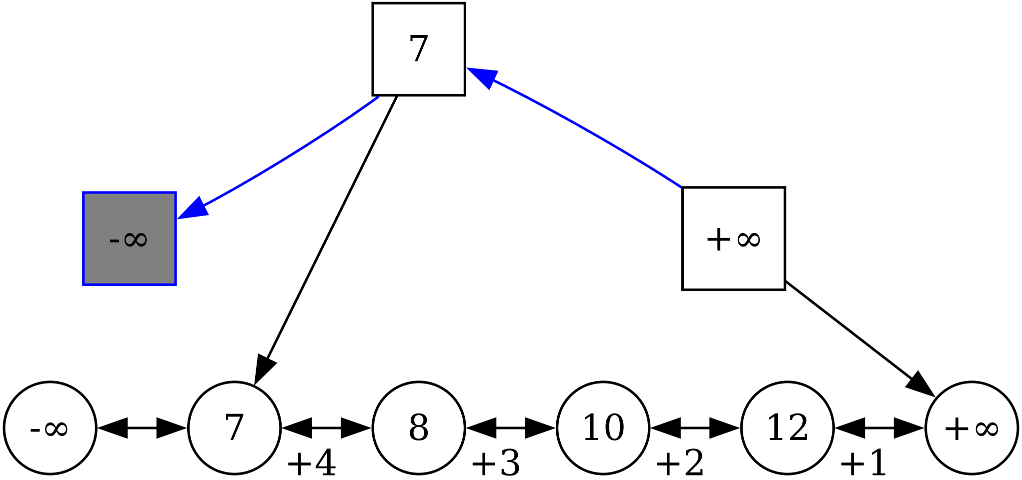
Insert 2 (doesn't find insertion location in $l = 4$ list nodes)



Traversal to $+\infty$, but the proper insertion location is not found within $l = 4$ comparisons through the list.

Henriksen's Algorithm

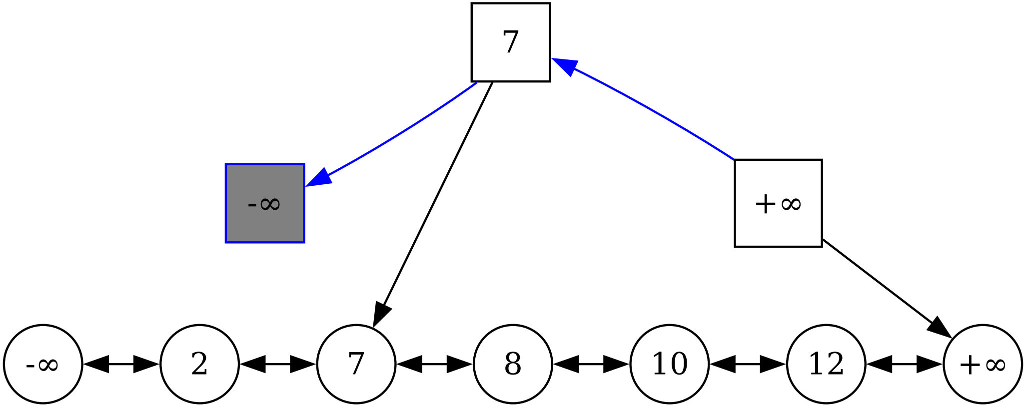
Insert 2 (requires a "pull" operation)



"Pull" the pointer belonging to the next smallest → node (in this case the root) to 7 in the linked list, and continue searching for the insertion location for 2.

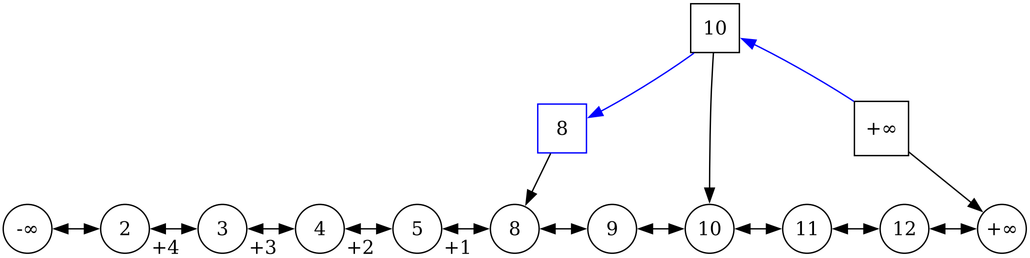
Henriksen's Algorithm

Insert 2 finished



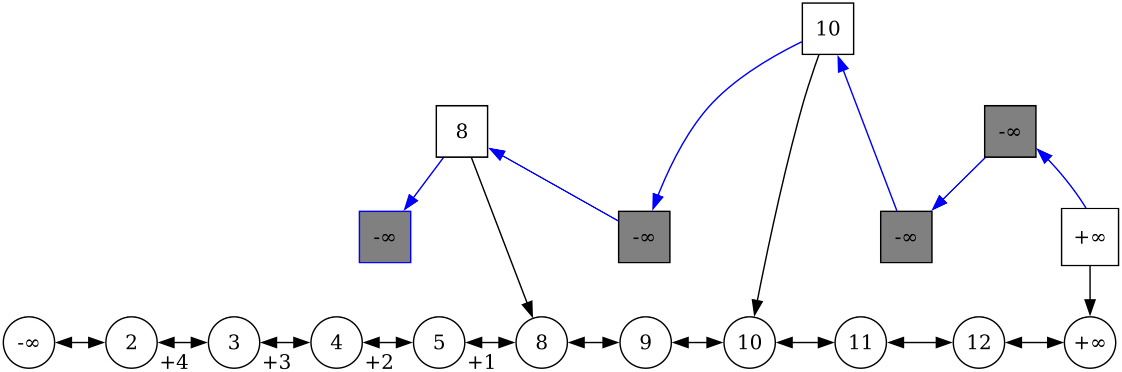
Henriksen's Algorithm

Insert 1 (requires a new tree level)



Henriksen's Algorithm

Insert 1 (new tree level initialized)

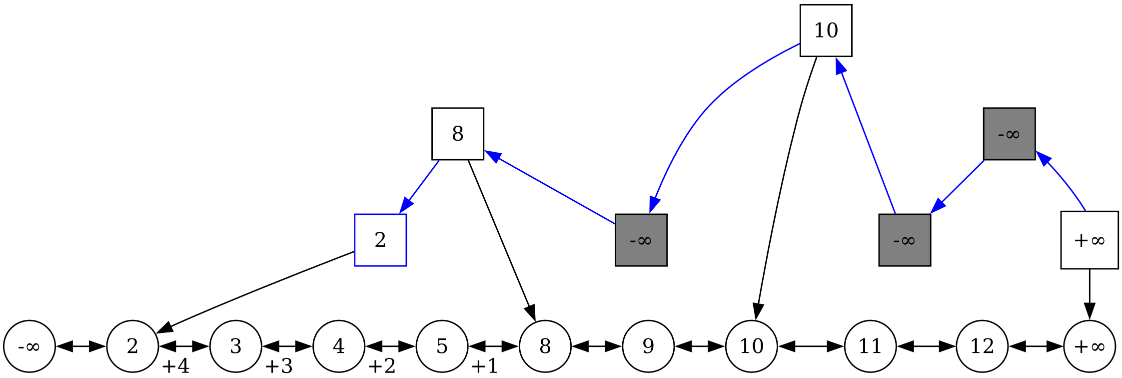


Notice the new next smallest → relationships.

All tree node pointers into the linked list remain the same **except for the** $+\infty$ **pointer**, which is migrated from the previous owner to the previous owner's right child (the new "greatest" node of the heap).

Henriksen's Algorithm

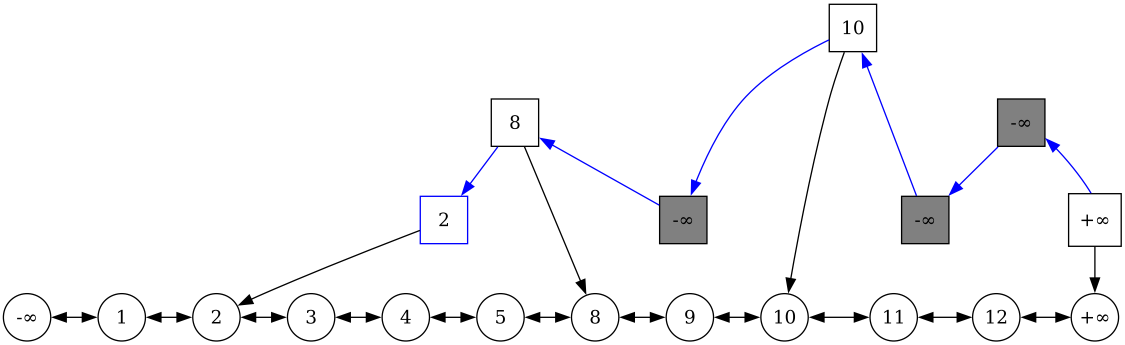
Insert 1 (pull with a newly available tree node)



Pull the next smallest → node from 8 (the tree node that pointed us to the list node where we began our search) to 2 in the linked list, continue searching for the proper insertion position for 1.

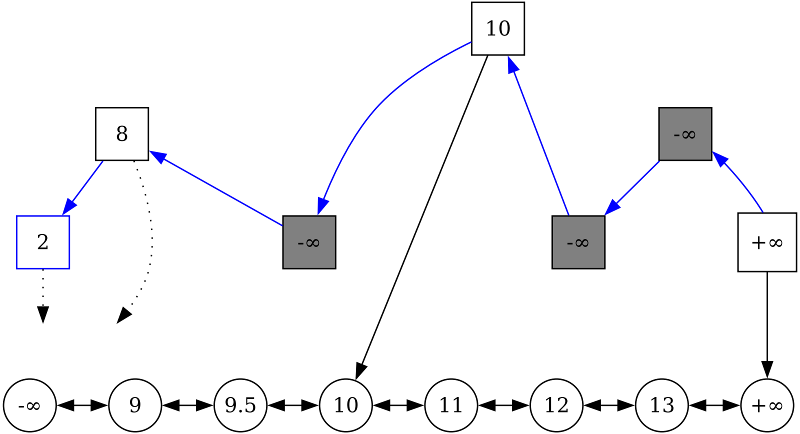
Henriksen's Algorithm

Insert 1 (finished)



Henriksen's Algorithm

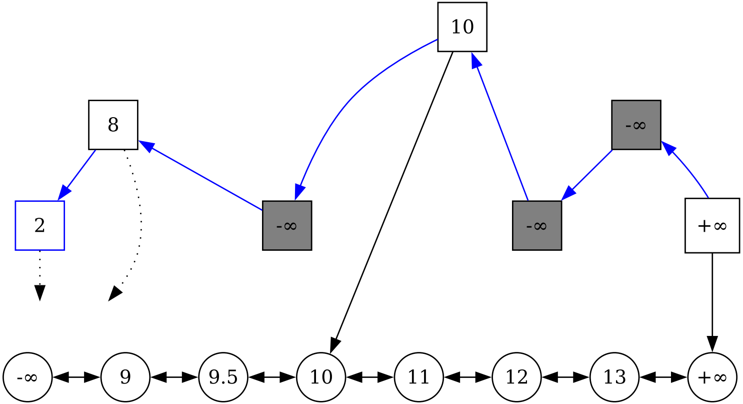
(After several events are processed) insert 8.2



The tree traversal will follow the path to 8, which is now a dangling pointer?!?
Back to the drawing board?

Henriksen's Algorithm

(After several events are processed) insert 8.2



No worries, all is well. If the event with activation time 8 is no longer in the list, **the simulation clock (t) is** > 8, recall:

$$followBranch = \begin{cases} leftChild & \text{if } node.at > t \text{ and } node.at > event.at \\ rightChild & \text{otherwise} \end{cases}$$

Important! Tree traversal always seeks (and remembers during traversal) the smallest *node.at* **larger than** *event.at*. If no such node is found, begin list search at $+\infty$.

Henriksen's Algorithm

(Amortized) enqueue $\Theta(\sqrt{n})$, dequeue $O(1)$,

isolated search and deletion $O(\log n)$

THE AMORTIZED COMPLEXITY OF HENRIKSEN'S ALGORITHM

JEFFREY H. KINGSTON

Department of Computer Science, The University of Iowa, Iowa City, Iowa 52242, U.S.A.

Abstract.

Henriksen's algorithm is a priority queue implementation that has been proposed for the event list found in discrete event simulations. It offers several practical advantages over heaps and tree structures.

Although individual insertions of $O(n)$ complexity can easily be demonstrated to exist, the "self-adjusting" nature of the data structure seems to ensure that these will be rare. For this reason, a better measure of total running time is the *amortized complexity*: the worst case over a sequence of operations, rather than for a single operation.

We show that Henriksen's algorithm has an amortized complexity of $\Theta(n^{1/2})$ per insertion, $O(1)$ per extract_min operation, and $O(\log n)$ for isolated deletions.