Examples of data structures that implement priority queues are: Fibonacci heaps, Red-black trees, Binary heaps.

Recall that a priority queue is a data structure for maintaining a set of elements, each with an associated key – priority. Typically the following operations are efficient:

- insert \( x \) – Insert \( x \).
- delete \( x \) – Remove \( x \).
- member \( x \) – Returns TRUE if \( x \) is in the dictionary, otherwise FALSE.
- minimum – Returns the smallest element in the dictionary.
- maximum – Returns the largest element in the dictionary.
- successor \( x \) – Return the smallest \( y \) in the dictionary such that \( y > x \).
- predecessor \( x \) – Return the largest \( y \) in the dictionary such that \( y < x \).

Recall, a priority queue is a data structure for maintaining a set of elements, each with an associated key – priority. Typically the following operations are efficient:

- insert \( x \) (insert element \( x \) which is associated with a key \( p \)).
- maximum or minimum (returns the element with the largest/smallest key).
- remove-maximum or remove-minimum (returns and removes the element with the largest/smallest key).

Examples of data structures that implement priority queues are:

- Binary heaps.
- Red-black trees.
- Fibonacci heaps.

These data structures base their decisions on comparing key values.

- Deterministic
  - Hashing does not allow operations like minimum and predecessor.
  - Also, we want to operate deterministically.
    - No randomisation!
    - No errors!

- Priority queues
  - We could use priority queues.
  - Recall, a priority queue is a data structure for maintaining a set of elements, each with an associated key – priority. Typically the following operations are efficient:
    - insert \( x \) (insert element \( x \) which is associated with a key \( p \)).
    - maximum or minimum (returns the element with the largest/smallest key).
    - remove-maximum or remove-minimum (returns and removes the element with the largest/smallest key).
  - Examples of data structures that implement priority queues are:
    - Binary heaps.
    - Red-black trees.
    - Fibonacci heaps.

- Fast & furious
  - Today we will show how to solve all of the operations

  - insert \( x \) – Insert \( x \).
    - Recall the meaning of \( \Omega \). If you do not remember, look it up! See for example CLRS, page 48.
  - delete \( x \) – Remove \( x \).
  - member \( x \) – Returns TRUE if \( x \) is in the dictionary, otherwise FALSE.
  - minimum – Returns the smallest element in the dictionary.
  - maximum – Returns the largest element in the dictionary.
  - successor \( x \) – Return the smallest \( y \) in the dictionary such that \( y > x \).
  - predecessor \( x \) – Return the largest \( y \) in the dictionary such that \( y < x \).

  in \( O(\log \log n) \) time (without randomisation) using \( O(u) \) space.
  - Recall that the universe \( U = \{0, \ldots, u - 1\} \).

  It can be shown that predecessor \( x \) must in fact take \( \Omega(\log \log u) \) time.

Computational model
- The solution will be given in the RAM model.
- The memory is organised as words of \( b \) bits.
- A word can be accessed in \( O(1) \) time.
- The number of bits \( b \) per word is \( \Theta(\log u) \), i.e. large enough to hold the value of any element in the universe.
- If it were not, we would not be able to address all the data (i.e. there would not be enough number of bits in a word to store a pointer to a cell in the memory that holds the value of an element from the universe).
- We also assume that standard word operations take \( O(1) \) time, e.g. addition, subtraction, multiplication, division, shifts and bitwise operations. (Think normal C programming.)
- In the RAM model we can beat the comparison-based lower bound of \( \Omega(n \log n) \) since we now operate on words.

Direct addressing
- Superimpose a binary tree over a bit array \( A \) of size \( u \).
- Element \( A[i] \) is set to 1 iff the key \( i \) is in the dictionary.
- An internal node is 1 if any of its children is 1.

Example
- Suppose for a second that we do not use recursion. Then we can use the previous constant height tree example.
- To address a word, only \( O(\log u) \) operations are needed. The memory is organised as words of \( b = \log u \), and so the RAM model can use the vEB tree to address a word.
- The memory is organised as words of \( b \), and so the RAM model can use the vEB tree to address a word.
- The memory is organised as words of \( b \), and so the RAM model can use the vEB tree to address a word.
- The memory is organised as words of \( b \), and so the RAM model can use the vEB tree to address a word.
- The memory is organised as words of \( b \), and so the RAM model can use the vEB tree to address a word.

Constant height tree
- The bit array \( A \) of size \( u \) is partitioned into \( \sqrt{u} \) blocks.
- Each block has size \( \sqrt{u} \) and are numbered from 0 to \( \sqrt{u} - 1 \).
- There is another bit array \( S \), the summary, of size \( \sqrt{u} \).
- Element \( S[i] \) is set if \( i \) is in \( A \) contains at least one 1.

Operation
- \( \text{insert}(x) \): \( O(1) \) time. Also set bit in \( S \) if necessary.
- \( \text{delete}(x) \): \( O(1) \) time. Check the block if it was the last 1.
- \( \text{member}(x) \): \( O(1) \) time. Read off the bit in \( A \).
- \( \text{minimum/maximum} \): \( O(1) \) time. Do min/max query on \( S \), then its block.
- \( \text{successor}(x) / \text{predecessor}(x) \): \( O(1) \) time. Search the block, then \( S \), then another block.

Example:
- \( u = 16 \) here, \( A \) contains at least one 1.
- \( S \) is set to 1 if the block \( i \) in \( A \) contains at least one 1.
- \( S \) is set to 1 if the block \( i \) in \( A \) contains at least one 1.
- \( S \) is set to 1 if the block \( i \) in \( A \) contains at least one 1.
- \( S \) is set to 1 if the block \( i \) in \( A \) contains at least one 1.

Van Emde Boas Trees
- We write vEB(\( u \)) to denote the vEB tree on a universe of size \( u \).
- vEB(\( u \)) has only one node that contains two variables:
  * min, containing the minimum element.
  * max, containing the maximum element.

Example:
- If \( \text{min} = \text{max} = 1 \) then only element 1 is in the dictionary.
- If \( \text{min} = 0 \) and \( \text{max} = 1 \) then both 0 and 1 have been inserted.
- If \( \text{min} = \text{max} = \text{null} \) then no element is in the dictionary.

Van Emde Boas Trees
- Recursively define the nodes of the van Emde Boas tree (vEB tree).
- The root of vEB(\( u \)) contains the following variables:
  * \( \text{min} \) and \( \text{max} \) (the min/max elements).
  * A pointer to the summary, which itself is a vEB(\( \sqrt{u} \)) tree.
  * An array of size \( \sqrt{u} \) that contains pointers to the \( \sqrt{u} \) blocks, each of which is a vEB(\( \sqrt{u} \)) tree.

Example:
- If \( u = 16 \) then only element 1 is in the dictionary.
- If \( u = 0 \) then both 0 and 1 have been inserted.
- If \( u = \text{null} \) then no element is in the dictionary.

Example:
- The root of vEB(\( u \)) contains:
- \( \text{min} = 2 \) \( \text{max} = 15 \)
- The root of vEB(16) contains:
- \( \text{min} = 2 \) \( \text{max} = 15 \)
- Summary pointer
- Block pointers
- \( S \) is set to 1 if the block \( i \) in \( A \) contains at least one 1.
- \( S \) is set to 1 if the block \( i \) in \( A \) contains at least one 1.
- \( S \) is set to 1 if the block \( i \) in \( A \) contains at least one 1.
- \( S \) is set to 1 if the block \( i \) in \( A \) contains at least one 1.
- \( S \) is set to 1 if the block \( i \) in \( A \) contains at least one 1.
- \( S \) is set to 1 if the block \( i \) in \( A \) contains at least one 1.
- \( S \) is set to 1 if the block \( i \) in \( A \) contains at least one 1.
- \( S \) is set to 1 if the block \( i \) in \( A \) contains at least one 1.
- \( S \) is set to 1 if the block \( i \) in \( A \) contains at least one 1.
- \( S \) is set to 1 if the block \( i \) in \( A \) contains at least one 1.
- \( S \) is set to 1 if the block \( i \) in \( A \) contains at least one 1.
- \( S \) is set to 1 if the block \( i \) in \( A \) contains at least one 1.
- \( S \) is set to 1 if the block \( i \) in \( A \) contains at least one 1.
- \( S \) is set to 1 if the block \( i \) in \( A \) contains at least one 1.
- \( S \) is set to 1 if the block \( i \) in \( A \) contains at least one 1.
STOP!!! There is a caveat!
The smallest element does not propagate down the tree.

The root of vEB(16) contains:
\[
\begin{array}{c}
\text{min} = 2 & \text{max} = 15 \\
\text{Summary pointer} & \text{Block pointers}
\end{array}
\]

Minimum element is already stored in min.

Operations on vEB trees

- minimum/maximum:
  Simply return the value of min/max. Done!
  This takes \(O(1)\) time. Easy peasy!

Indexing elements – a technicality

Let \(x\) be an element.

Define
\[
\begin{align*}
\text{high}(x) &= \left\lfloor \frac{x}{\sqrt{u}} \right\rfloor \\
\text{low}(x) &= x \mod \sqrt{u}
\end{align*}
\]

- high\((x)\) is the block that contains \(x\).
- low\((x)\) is the relative position of \(x\) within this block.
- Both high\((x)\) and low\((x)\) can be computed in constant time.

Indexing elements – a technicality

STOP!!! There is a caveat!
The smallest element does not propagate down the tree.

The root of vEB(16) contains:
\[
\begin{array}{c}
\text{min} = 2 & \text{max} = 15 \\
\text{Summary pointer} & \text{Block pointers}
\end{array}
\]

The smallest element does not propagate.

Operations on vEB trees

- \(\text{predecessor}(x)\):
  1. If \(u = 2\), check min and max and return appropriately. Done!
  2. If \(x > \text{max}\), return \(\text{max}\). Done!
  3. Let \(y = \text{high}(x)\) be the block containing \(x\).
     If the min element in \(y\) is less than \(x\) (and not NULL), run \(\text{predecessor}\) on \(y\). Done!
  4. a) If not, run \(\text{predecessor}\) on the summary vEB.
     b) If say block \(z\) is returned, return the max element of \(z\). Done!
     c) If \text{NULL} is return (no smaller element), return \text{NULL}. Done!

  - Time analysis: suppose the running time for predecessor is \(T(u)\).
    - Steps 1 and 2 take \(O(1)\) time.
    - Step 3 takes \(T(\sqrt{u})\) time.
    - Step 4a takes \(T(\sqrt{u})\) time.
    - Steps 4b and 4c take \(O(1)\) time.

Thus, total time is \(T(u) \leq T(\sqrt{u}) + O(1)\).

Solving this recurrence gives \(T(u) = O(\log \log u)\).
Operations on vEB trees

- **insert**\(^{(x)}\):
  1. If the dictionary is empty, set min and max to \(x\). Done!
  2. If \(x < \text{min}\), swap \(x\) and \(\text{min}\). Now \(x\) is second smallest.
  3. Let \(y = \text{high}(x)\) be the block containing \(x\).
     - If block \(y\) is empty, insert \(y\) into the summary and insert \(x\) in \(y\).
     - Otherwise, only insert \(x\) into block \(y\) (no need to touch the summary).
  4. Update max is necessary, i.e. if \(x > \text{max}\) then \(\text{max} \leftarrow x\). Done!

**Time analysis:** suppose the running time for insert is \(T(u)\).

- Steps 1 takes \(O(1)\) time.
- Step 2 takes \(O(1)\) time.
- Step 3 takes \(T(\sqrt[3]{u})\) time (inserting \(y\) into the summary). Inserting \(x\) into the empty \(y\) takes \(O(1)\) time because of step 1.
- Step 4 takes \(T(\sqrt[3]{u})\) time.
- Steps 5 takes \(O(1)\) time.

Again, \(T(u) \leq T(\sqrt[3]{u}) + O(1)\), implying \(T(u) = O(\log \log u)\).

Space usage

- The other operations are left as an exercise.

- How much space does the vEB tree use?
  - Let \(Z(u)\) be the size of a vEB tree with universe of size \(u\).
  - The root contains \(\sqrt[3]{u} + 1 = O(\sqrt[3]{u})\) pointers.
  - Each tree has size (roughly) \(Z(\sqrt[3]{u})\).
  - Thus, total size is \(Z(u) \leq \sqrt[3]{u} \cdot Z(\sqrt[3]{u}) + O(1)\).
  - Solving this recurrence gives \(Z(u) = O(u)\).

That is, the space usage of the whole vEB tree is linear in \(u\).