Semi-Heap and Its Applications in Tournament Ranking

Jie Wu

Department of Computer Science and Engineering

Florida Atlantic University

Boca Raton, FL 33431

jie@cse.fau.edu

September 21, 2001

1 Introduction

Two different worlds

- **Hardyism**

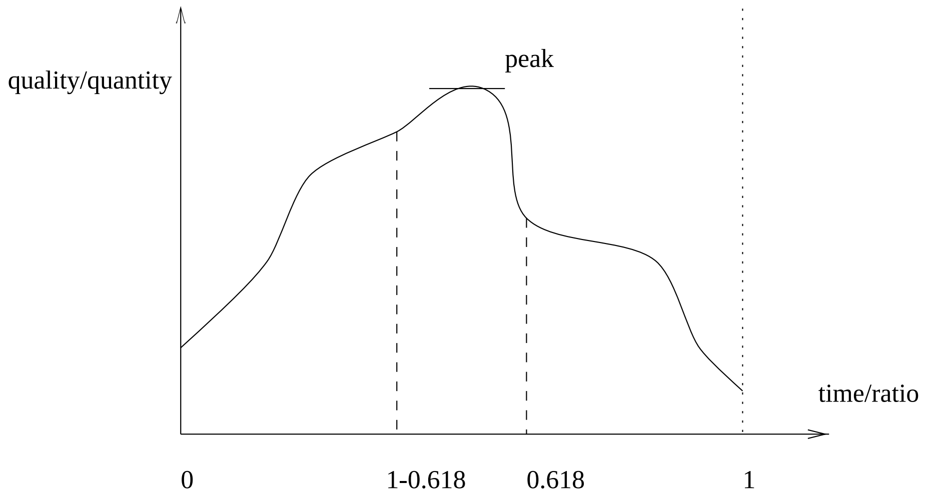
  Utility as a goal is inferior to elegance and profundity.

- **Maoism**

  Scientific research should serve proletarian polities, ..., and be intergrated with productive labor.

Golden-ratio-based Search:

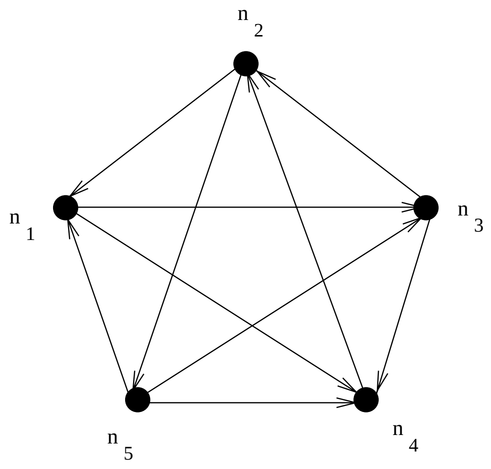
Golden ratio: $\phi = \frac{\sqrt{5}-1}{2} = 0.618...$



Questions:

- Why golden-ratio-based search?
- Golden-ratio-based search or binary-tree-based search?

- **Tournament**: n players where every possible pair of players plays one game to decide the winner (and the loser) between them.
- **Graph representation**: A directed graph with a complete underlying graph.



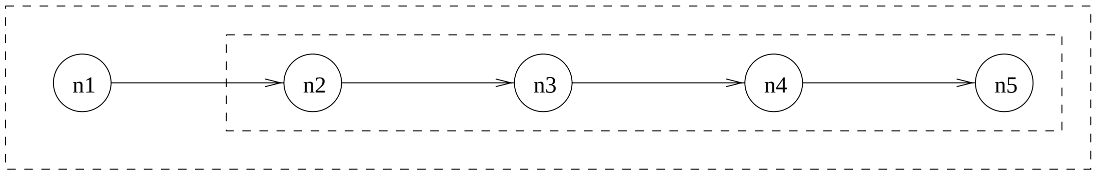
- **Hamiltonian path** (also called **generalized sorted sequence**):

$$n_3 \succ n_4 \succ n_2 \succ n_5 \succ n_1$$

- Lower bound: $\Theta(n \log n)$
- Sorting algorithms: bubble sort, binary insertion sort, merge sort
- We extend a heapsort algorithm using a **semi-heap** and then generalize it to a cost-optimal parallel algorithm under the EREW PRAM model with $\Theta(n)$ run time using $\Theta(\log n)$ processors.

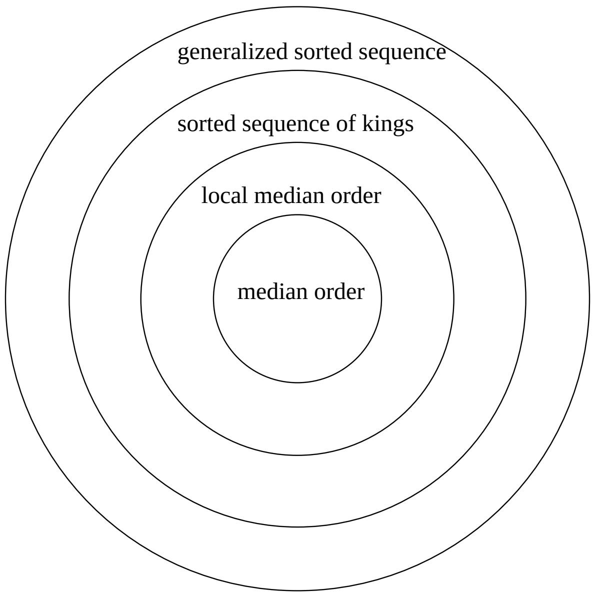
Tournament Ranking Problem:

- generalized sorted sequence ($\Theta(n \log n)$)
- sorted sequence of kings (Wu 2000, conjectured to be $\Theta(n^2)$)
- local median order ($O(n^4)$)
- median order (NP-complete)



king: other players are beaten by the king directly or indirectly (via a third player).

median order: ranking of players with a minimum number of total **upsets**.



2 Preliminaries

Tournament

Existence of a Hamiltonian path in any tournament:

Preposition: *Consider a set N ($|N| = n$) with any two elements n_i and n_j, either $n_i \succ n_j$ or $n_j \succ n_i$. Elements in N can be arranged in a linear order*

$$n_1^{'} \succ n_2^{'} \succ ... \succ n_{n-1}^{'} \succ n_n^{'}$$

- Assume that the preposition holds for $n = k$:

$$n_1^{'} \succ n_2^{'} \succ ... \succ n_k^{'}$$

- When $n = k + 1$, we insert the $(k + 1)$th element $n_{k+1}^{'}$ in front of $n_i^{'}$, where i is the *smallest index* such that $n_{k+1}^{'} \succ n_i^{'}$:

$$n_1^{'} \succ n_2^{'} \succ ... \succ n_{k+1}^{'} \succ n_i^{'} ... \succ n_k^{'}$$

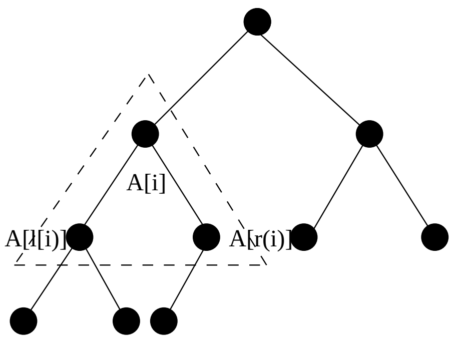
- If such an index i does not exist, $n_{k+1}^{'}$ is placed as the last element:

$$n_1^{'} \succ n_2^{'} \succ ... \succ n_k^{'} \succ n_{k+1}^{'}$$

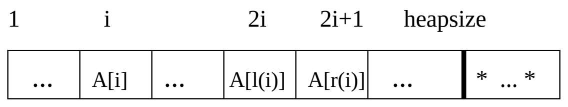
Heap

- Heap is an array A that can be viewed as a **complete binary tree**.
- The left child of $A[i]$ is $A[l(i)] = A[2i]$ and the right child of $A[i]$ is $A[r(i)] = A[2i+1]$.
- **Heap property**: For every node i other than the root:

$$A[Parent(i)] \geq A[i]$$

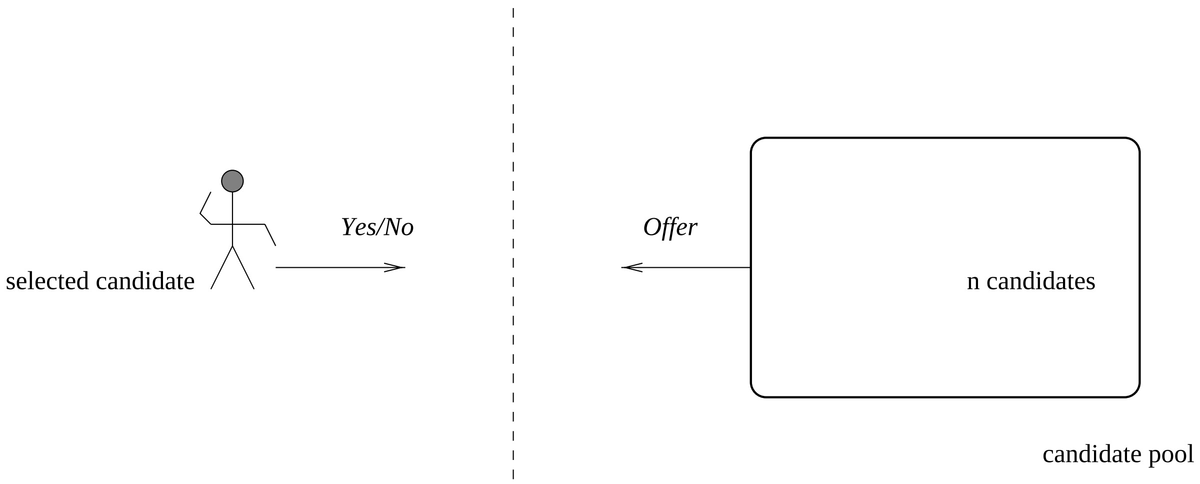


(a)



(b)

Faculty Recruting Process:



A k-round selection process

Cost function:

Type	Random	Sorted	Heap
Construct	$\Theta(1)$	$\Theta(n \log n)$	$\Theta(n)$
Select	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Maintain	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$

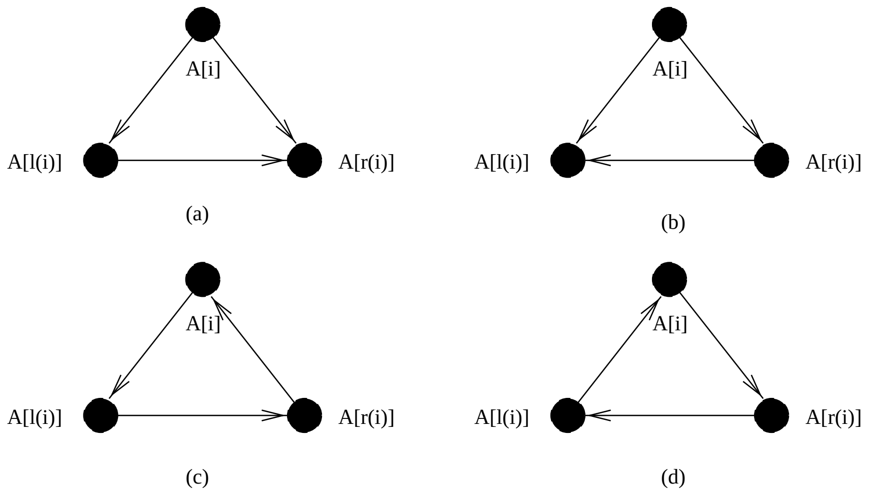
Overall cost:

Type	Random	Sorted	Heap
$k = \Theta(1)$	$\Theta(n)$	$\Theta(n \log n)$	$\Theta(n)$
$k = \Theta(n)$	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$
$k = \Theta(n/\log n)$	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n)$

3 Semi-Heap

Definition 1: *$n_1 = \max_{\succ}\{n_1, n_2, n_3\}$ if both $n_2 = \max\{n_1, n_2, n_3\}$ and $n_3 = \max\{n_1, n_2, n_3\}$ are false.*

Four possible configurations of a triangle in a semi-heap.



Definition 2: *A semi-heap for a given intransitive total order $\succ$ is a complete binary tree. For every node $n^{'}$ in the tree, $n^{'} = \max_{\succ}\{n^{'}, L(n^{'}), R(n^{'})\}$.*

Construct a semi-heap from a random array:

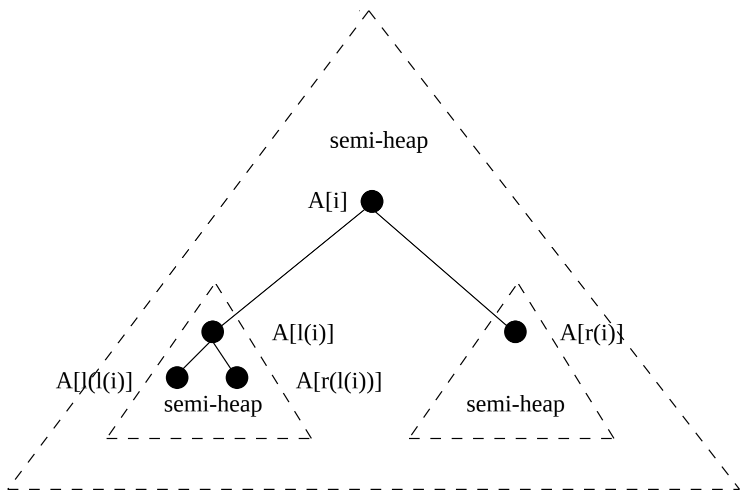
- SEMI-HEAPIFY(A, i) constructs a semi-heap rooted at $A[i]$, provided that binary trees rooted at $A[l(i)]$ and $A[r(i)]$ are semi-heaps.
  (Its cost is $\Theta(\log n)$, where $n = \mathit{heapsize}$.)

- BUILD-SEMI-HEAP(A) uses the procedure SEMI-HEAPIFY in a bottom-up manner to convert an arbitrary array A into a semi-heap.
  (Its cost is $\Theta(n)$)

```
SEMI-HEAPIFY($A, i$)
1  if $A[i] \neq \max_{\succ}\{A[i], A[l(i)], A[r(i)]\}$
2     then find $winner$ such that
              $A[winner] \longleftarrow \max\{A[i], A[l(i)], A[r(i)]\}$
3          exchange $A[i] \longleftrightarrow A[winner]$
4          SEMI-HEAPIFY($A, winner$)
```

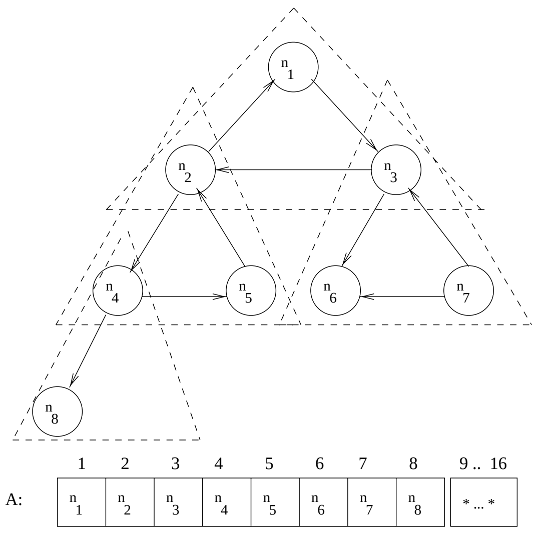
```
BUILD-SEMI-HEAP($A$)
1  for  $i \longleftarrow \lfloor \frac{heapsize}{2} \rfloor$ downto 1
2        do SEMI-HEAPIFY($A, i$)
```

The description of the SEMI-HEAPIFY algorithm:

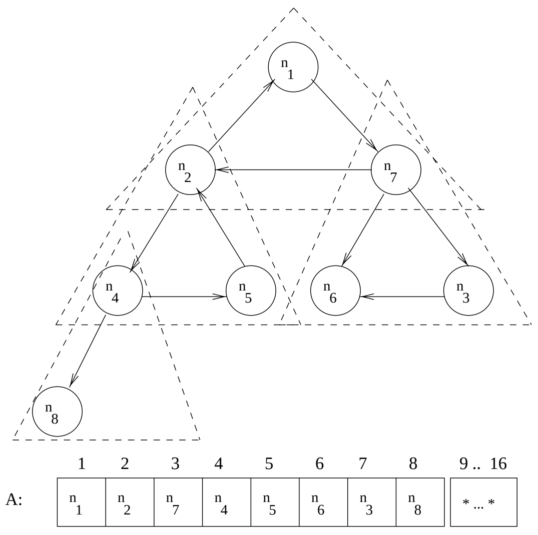


Theorem 1: *BUILD-SEMI-HEAP constructs a semi-heap for any given complete binary tree.*

An example of using BUILD-SEMI-HEAP:



(a)

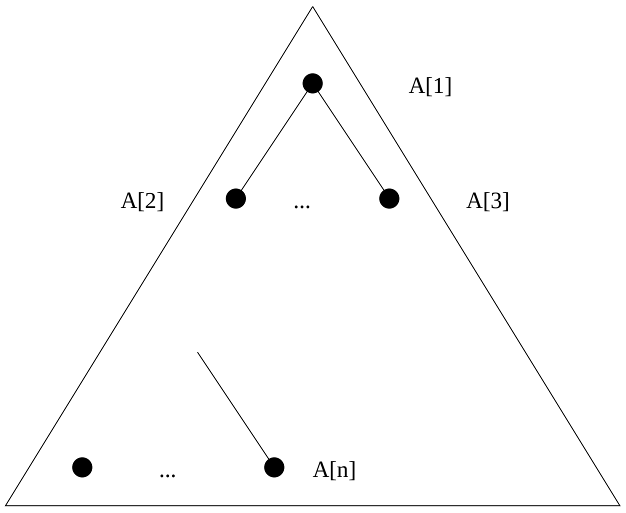


(b)

4 Generalized Sorting Using Semi-Heap

Why the traditional heapsort cannot be used?

- With the transitive property, root $A[1]$ "beats" all the other "players".
- When the root is discarded, it is replaced by the last element $A[n]$ in the heap.
- Then the heap is reconstructed by pushing $A[n]$ down in the heap if necessary so that the new root is the maximum element among the remaining ones.
- In a semi-heap, the following situation may occur: $A[n]$ "beats" all $A[1]$, $A[2]$, and $A[3]$.



Generalized sorting using semi-heap

- Generalized sorting is done through SEMI-HEAP-SORT by repeatly printing and removing the root of the binary tree (which is initially a semi-heap).
- The root is replaced by either its left child or right child through REPLACE.
- The selected child is replaced by one of its children. The process continues until a leaf node is reached and the entry for the leaf node is replaced by $*$.

- REPLACE(A, i) repeatly replaces a node (starting from the root) by either its leftchild or rightchild until the current node is a leave node. (Its cost is bounded the height of the original semi-heap, $\Theta(\log n)$).
- SEMI-HEAP-SORT repeatly prints and removes the root of the binary tree (which is initially a semi-heap).
  (Its cost is $\Theta(n \log n)$.)

```
REPLACE$(A, i)$
  1  if  $(A[l(i)] = *) \wedge (A[r(i)] = *)$
  2     then $A[i] \longleftarrow *$
  3     else  if $(A[i] \succ A[l(i)]) \wedge (A[l(i)] \succ A[r(i)])$
  4               then $A[i] \longleftarrow A[l(i)]$
  5                    REPLACE$(A, l[i])$
  6               else  $A[i] \longleftarrow A[r(i)]$
  7                    REPLACE$(A, r[i])$
```

```
SEMI-HEAP-SORT$(A)$
  1  BUILD-SEMI-HEAP(A)
  2  while $(A[l(1)] \neq *) \vee (A[r(1)] \neq *)$
  3          do  print$(A[1])$
  4              REPLACE$(A$, 1$)$
  5  print$(A[1])$
```

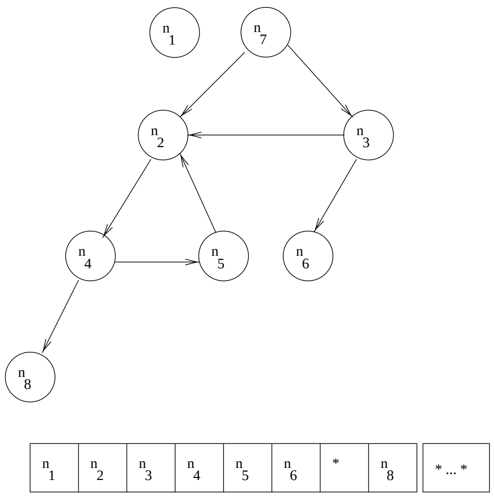
Theorem 2: *For any given semi-heap, SEMI-HEAP-SORT generates a generalized sorted sequence.*

Tournament representation:

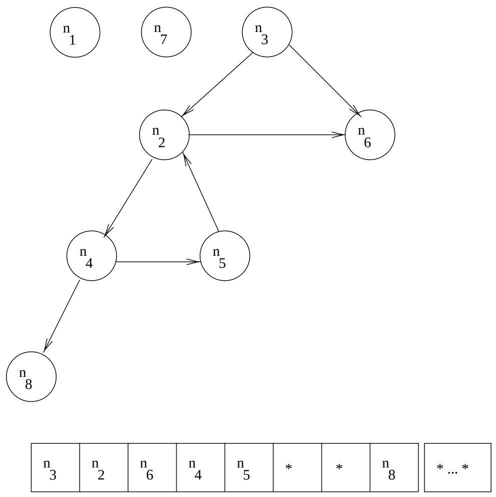
- A tournament is represented by an $n \times n$ matrix M.
- $M[i, j] = 1$ if n_i beats n_j (i.e., $n_i \succ n_j$).
- $M[i, j] = 0$ if n_i is beaten by n_j (i.e., $n_j \succ n_i$).
- $M[i, i] = -$ represents an impossible situation.

$$M = \begin{pmatrix} - & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & - & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & - & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & - & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & - & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & - & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & - & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & - \end{pmatrix}$$

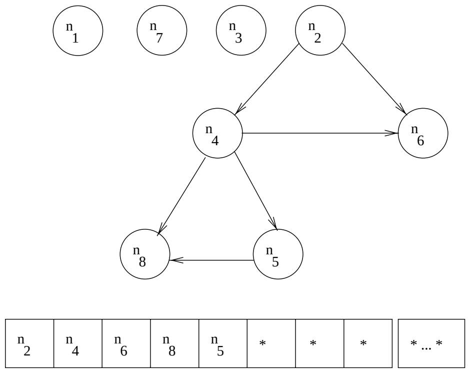
A step-by-step application of REPLACE(A, i):


n_1 n_7 n_2 n_3 n_4 n_5 n_6 n_8
n_1 n_2 n_3 n_4 n_5 n_6 * n_8 * ... *

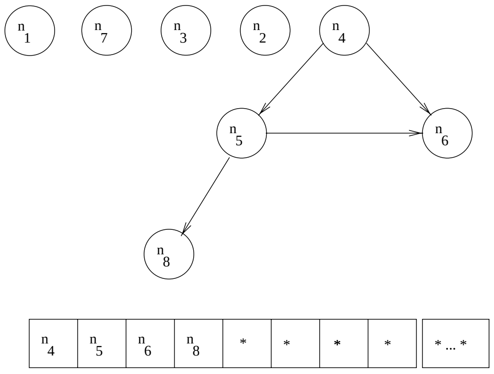
(a)


n_1 n_7 n_3 n_2 n_6 n_4 n_5 n_8
n_3 n_2 n_6 n_4 n_5 * * n_8 * ... *

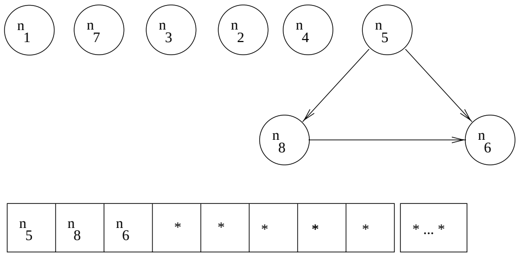
(b)


n_1 n_7 n_3 n_2 n_4 n_6 n_8 n_5
n_2 n_4 n_6 n_8 n_5 * * * * ... *

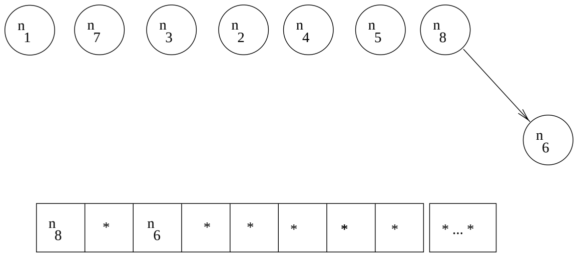
(c)


n_1 n_7 n_3 n_2 n_4 n_5 n_6 n_8
n_4 n_5 n_6 n_8 * * * * * ... *

(d)


n_1 n_7 n_3 n_2 n_4 n_5 n_8 n_6
n_5 n_8 n_6 * * * * * * ... *

(e)


n_1 n_7 n_3 n_2 n_4 n_5 n_8 n_6
n_8 * n_6 * * * * * * ... *

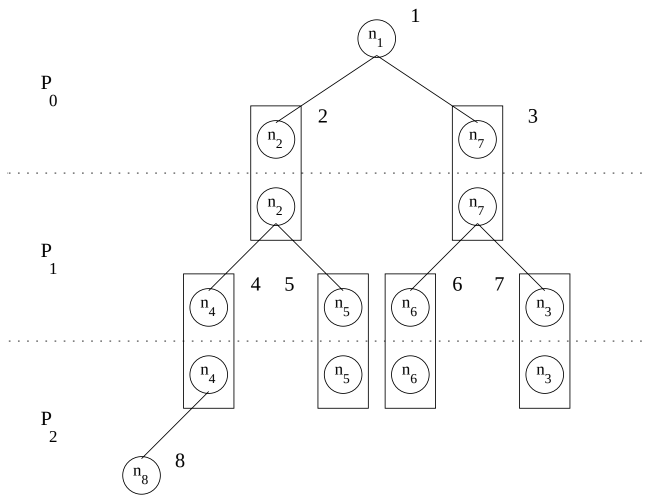
(f)

5 Parallel Generalized Sorting Using Semi-Heap

A cost-optimal parallel algorithm

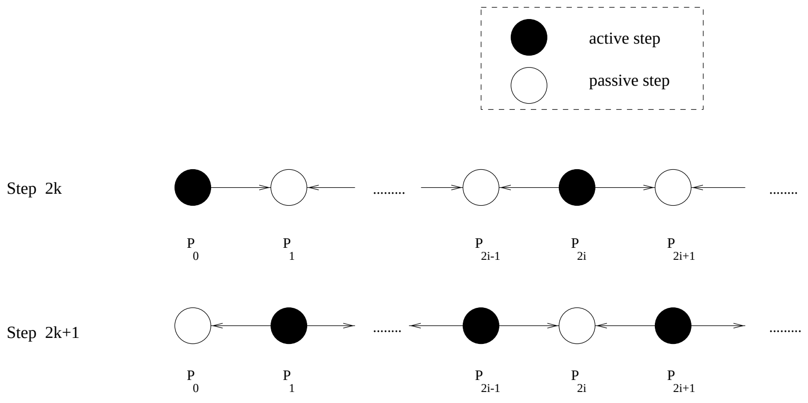
- A sorting algorithm is **cost-optimal** if the product of run time and the number of processors is $\Theta(n \log n)$.
- REPLACE$(A, 1)$ is pipelined level to level and this statement is called at every other step (since each node is shared by two processors at adjacent level, an idle step is inserted between two calls).
- The run time of SEMI-HEAP-SORT is reduced to $\Theta(n)$ with $\Theta(\log n)$ processors.
- This parallel algorithm runs on the CREW PRAM model, but can be easily modified to the EREW PRAM model without additional cost.

From CREW PRAM to EREW PRAM: resolve **memory access conflict**



The **network model**: a linear array of processors P_0, P_1, P_2, ... P_h, where $h = \lceil \log n \rceil$.

Active and **passive** steps:



- At an even step, processors P_0, P_2, P_4, ... take the active step and processors P_1, P_3, P_5, ... take the passive step.
- The role of active and passive among these processors exchanges in the next step.

Theorem 3: *The proposed parallel implementation is cost-optimal with a run time of $\Theta(n)$ using $\Theta(\log n)$ processors.*

P_0 at an active step (starts from step 0):

1. Prints root $A[1]$.
2. If both child nodes are $*$, $A[1]$ is replaced by $*$ and then P_0 sends a termination signal to P_1 and stops.

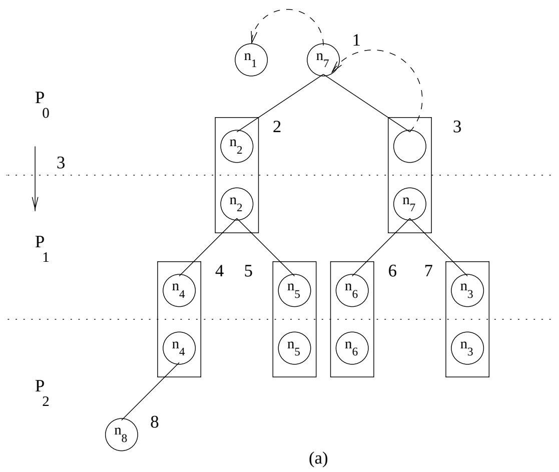
   If at least one child node is not $*$, replaces $A[1]$ by one of two child nodes, $A[2]$ and $A[3]$, following the rule in REPLACE.

   - If A[2] is selected, P_0 sends $id = 2$ to processor P_1; otherwise, $id = 3$ is sent.
   - In the next step (a passive step), P_0 receives $(id, replacement)$ from P_1, and then, performs the update $A[id] := replacement$.

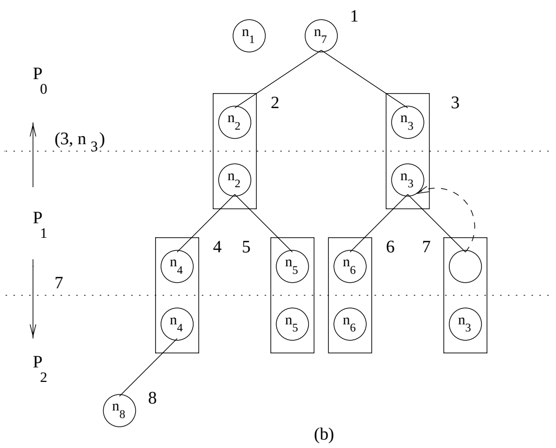
P_i, $i \neq 0$, at a passive step:

- If P_i receives $(id, replacement)$ from P_{i+1}, it performs the update $A[id] := replacement$.
- If P_i receives signal $id = j$ from P_{i-1}, it performs the following activities in next active step:
    1. If both children are $*$, $A[j]$ is replaced by $*$; otherwise, $A[j]$ is replaced by either $A[2j]$ or $A[2j+1]$ based the replacement rule.
    2. Send $(j, A[j])$ to P_{i-1}.
    3. If either $A[2j]$ or $A[2j+1]$ is selected to replace $A[j]$, the corresponding id ($2j$ or $2j+1$) is sent to P_{i+1}, provided P_i is not the last processor (i.e., $i \neq h-1$); otherwise, the selected element is replaced by $*$.
- If P_i receives the termination signal, it forwards the termination signal to the next processor P_{i+1} (if it exists) in the next active step, and then, P_i stops.

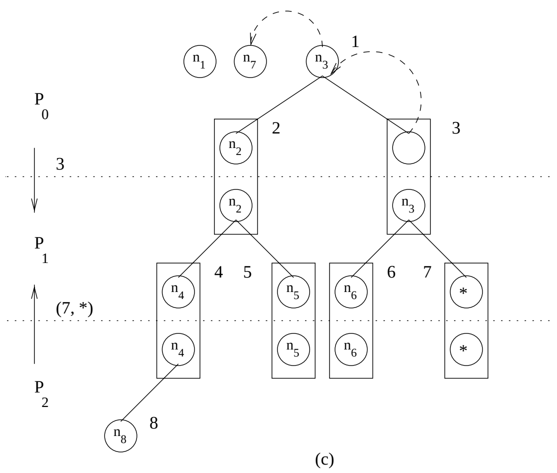
A step-by-step illustration:



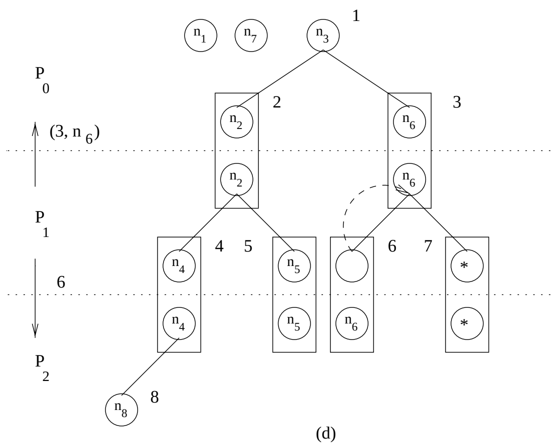
(a)



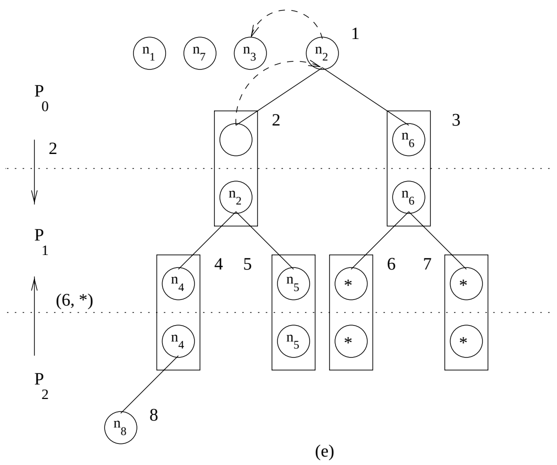
(b)



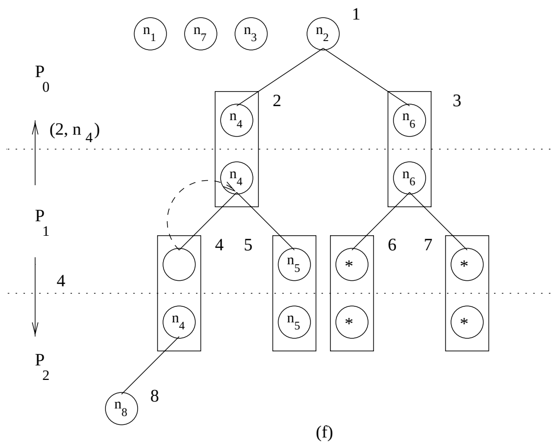
(c)



(d)



(e)



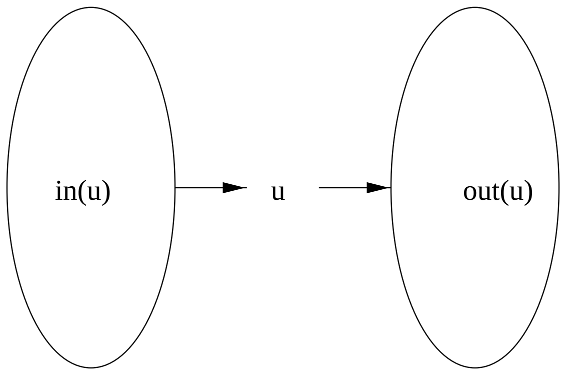
(f)

6 Other Results

Sorted Sequence of Kings:

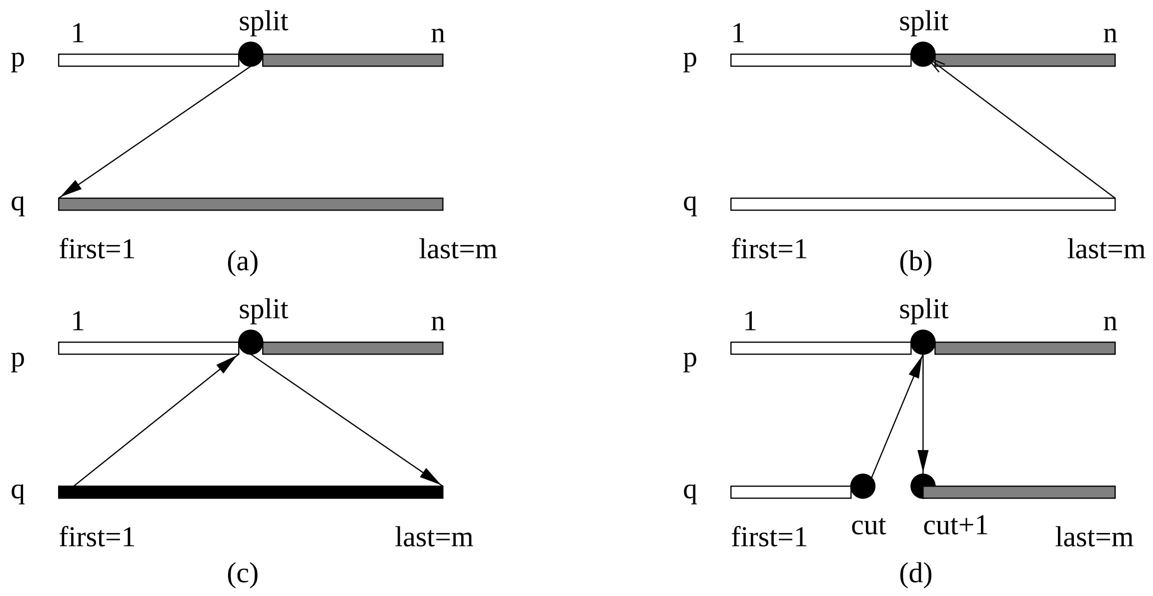
Quicksort

- $in(u)$: a set of players that beat u.
- $out(u)$: a set of players that are beaten by u.



<u>Parallel Merge</u>:

An EREW PRAM model with running time $O(\log^2 n)$ using $O(n/\log^2 n)$ processors.



7 Conclusions

- A data structure called semi-heap.
- An optimal solution to the generalized sorting problem.
- A cost-optimal EREW PRAM algorithm with $\Theta(n)$ in run time using $\Theta(\log n)$ processors.
- An implementation of the proposed parallel algorithm under the network model using a linear array of processors.