CS 240 Programming Assignment Phase 2

Winter Semester 2024-2025

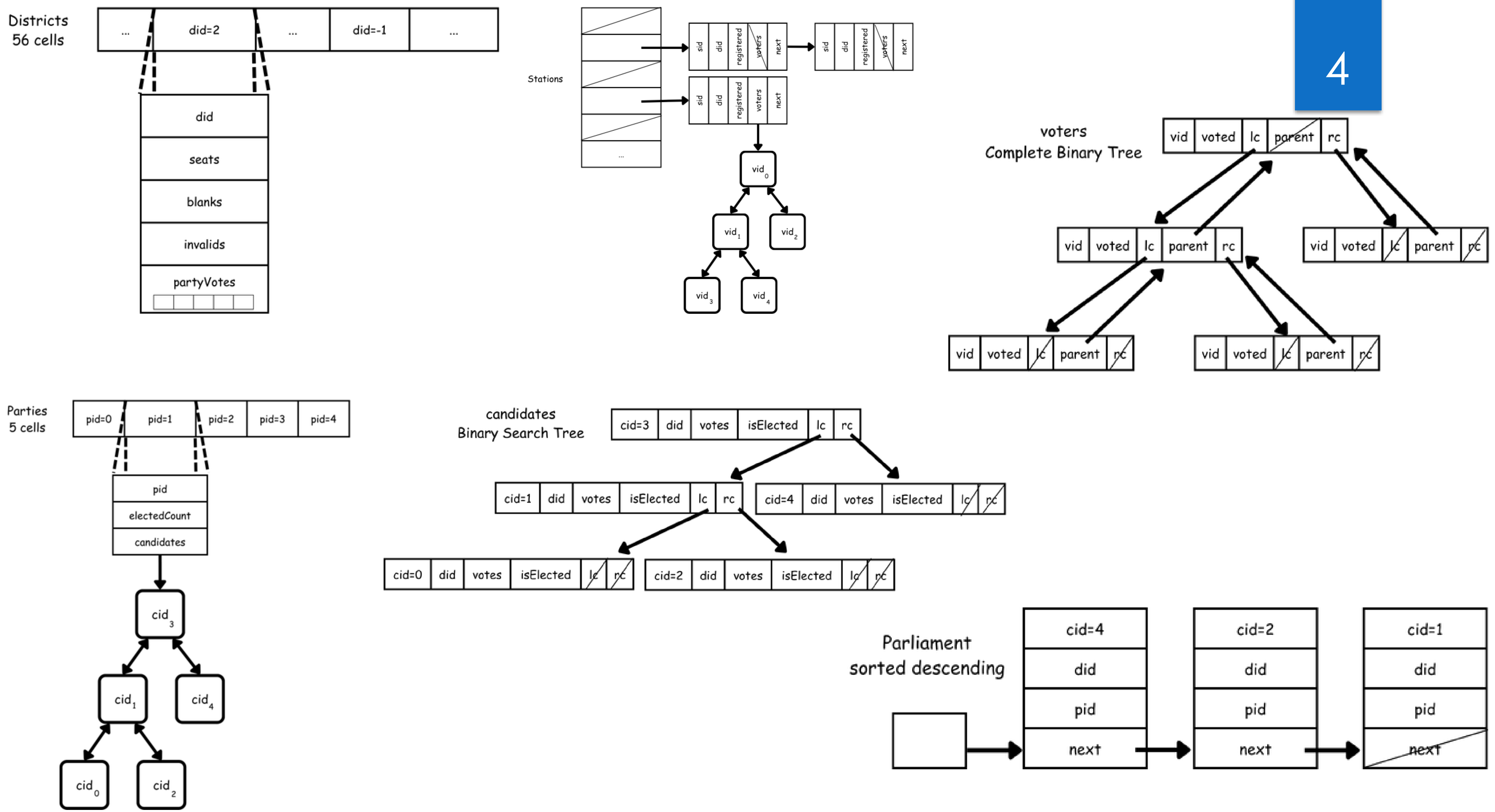
Myron Tsatsarakis – myrontsa@csd.uoc.gr

3 December 2024

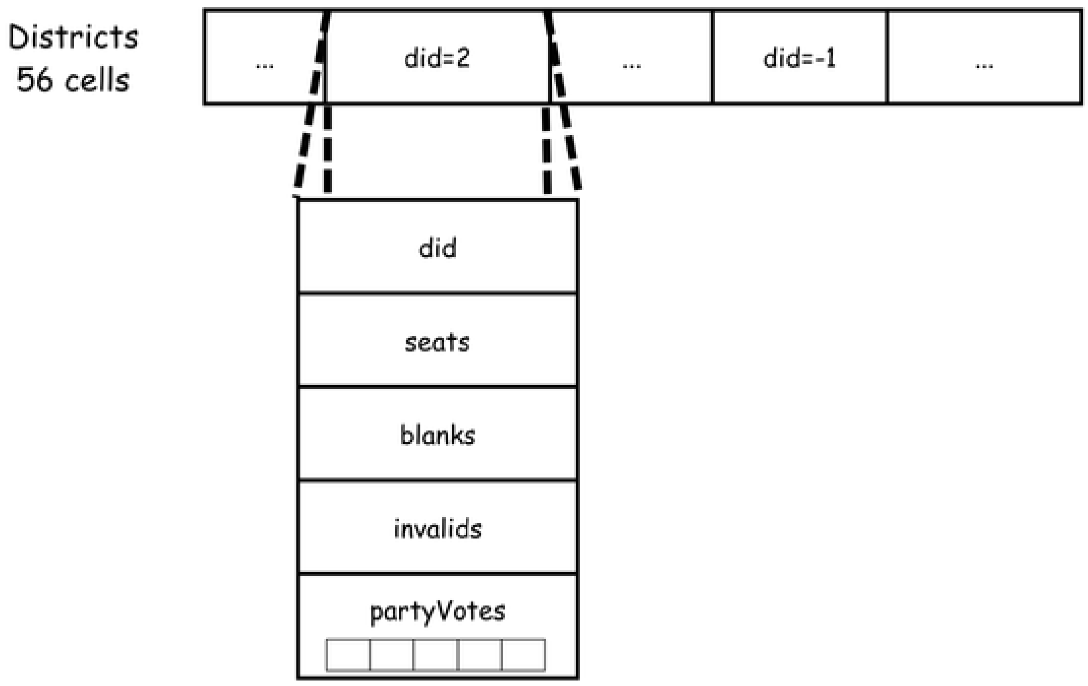
Concept

- **Main Idea** loosely simulate Greek elections
- 56 Districts in global array
- 5 Parties in global array
  - Candidates registered per party (candidates Binary Search Tree)
  - Each candidate stores his/her district id
- Stations in global Hashtable
  - Each station stores its district id
  - Voters registered per station
- Final formed parliament → Elected candidates from all parties

Design & Data Structures

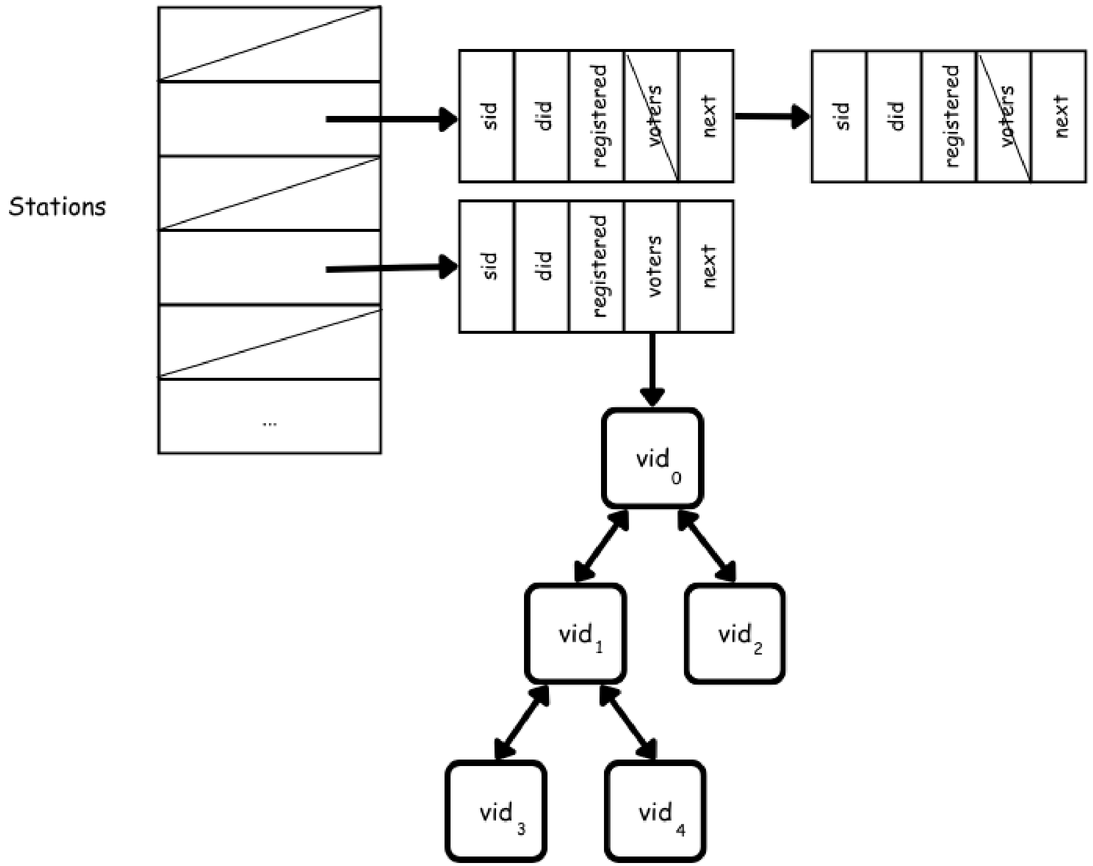

Districts
56 cells
...
did=2
...
did=-1
...
did
seats
blanks
invalids
partyVotes
Stations
sid
did
registered
voters
next
vid 0
vid 1
vid 2
vid 3
vid 4
voters
Complete Binary Tree
vid
voted
lc
parent
rc
Parties
5 cells
pid=0
pid=1
pid=2
pid=3
pid=4
pid
electedCount
candidates
cid 3
cid 1
cid 4
cid 0
cid 2
candidates
Binary Search Tree
cid=3
did
votes
isElected
lc
rc
cid=1
cid=4
cid=0
cid=2
Parliament
sorted descending
cid=4
cid=2
cid=1
did
pid
next

Districts Array



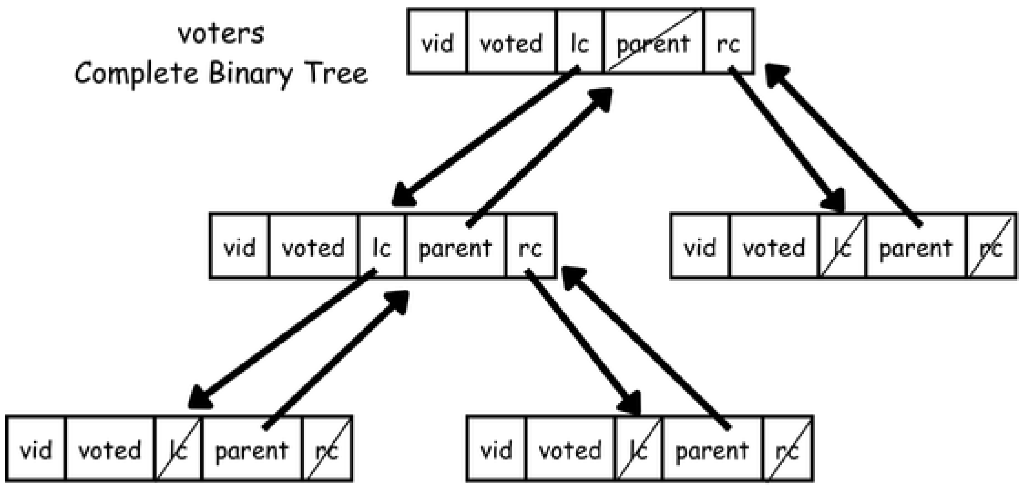
- **Districts**
  - array of 56 cells of type District
- Type District
  - **did** unique district id
  - **seats** number of seats to be distributed
  - **blanks** counter of blank votes
  - **invalids** counter of invalid votes
  - **partyVotes** array of 5 cells, holding counters of total valid votes for each party

Stations Hashtable



- **Stations**
  - Hashtable with capacity of your choosing
  - Dynamically allocated array of ordered chains
  - Each chain contains elements of type Station
- Type Station
  - **sid** unique station id
  - **did** district id where station is located
  - **registered** counter of voters registered to the station
  - **voters** tree containing voters registered to the station
  - **next** pointer to next station node in the chain

Voters Complete Binary Tree



- **Complete, Unordered, Doubly - Linked, Binary Tree**
- contains elements of type Voter
- **voters** (member of Station)
  - pointer to the root node
- Type Voter
  - **vid** unique voter id
  - **voted** boolean indicating if voter has cast a vote
  - **parent** pointer to parent node
  - **lc** pointer to left child node
  - **rc** pointer to right child node

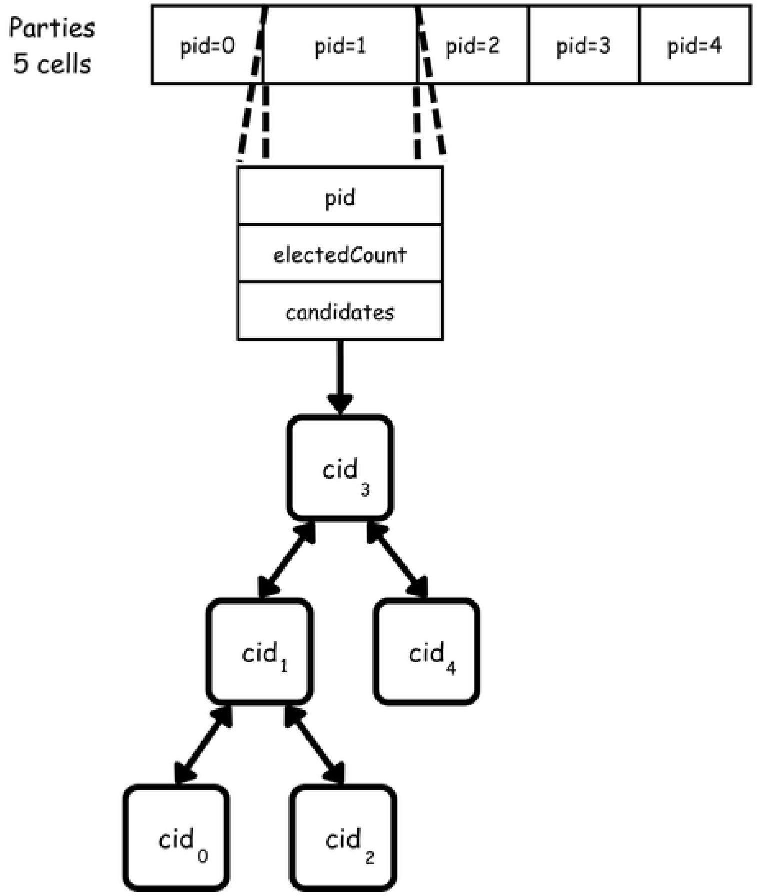
Universal Hashing

- **Main Idea** Define a collection of hash functions. Pick one at random to use during a single program execution.
- Let **m** be the hashtable capacity
- Let **K** be the max possible key contained in the hashtable
- Pick number **p** from Primes array, where **p** > **K**
  - Primes array is a global, already initialized in the code
- Pick random number **a** in the range [1, **p**)
- Pick random number **b** in the range [0, **p**)
- Define the hash function for this program execution
  - int Hash(int key) { return ((**a** * key + **b**) % **p**) % **m**; }
- In the test files we provide additional input
  - **MaxSid** the maximum station id contained in the testfile
  - Utilize it
- Set a constant seed to the random function, **for easier debugging**
- **Study the slide sections on Universal Hashing**

Picking Hashtable Capacity

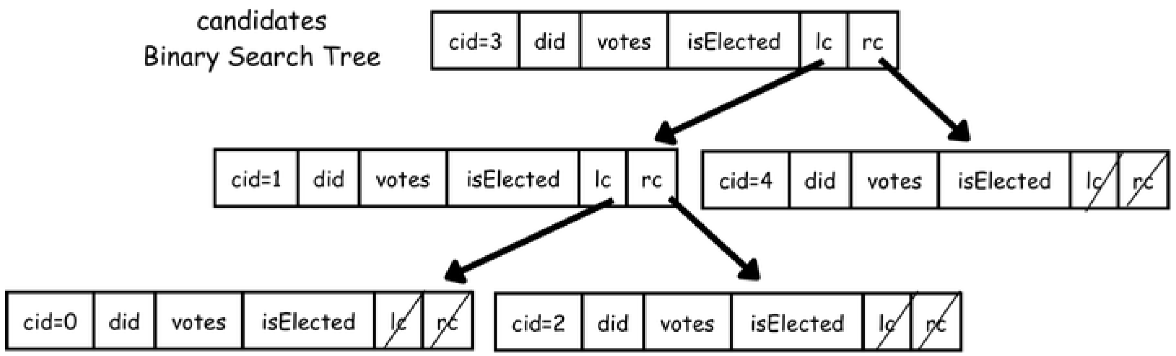
- Let **n** be the total number of elements to be inserted in the hashtable
- Let **m** be the capacity of the hashtable. How to pick a good value for **m**?
- Load factor defined as **a** == **n** / **m**
- When **a <= 1**, then search performance is optimal (for an ideal hash function)
- **a** == **n** / **m** ➔ **m** == **n** / **a** ➔ for **a <= 1**, **m >= n**
- So, we need to pick a capacity equal or greater to the number of elements
- Universal hashing does not pose any additional restrictions on capacity
  - **m** can be a prime number from Primes array
  - **m** can be an odd number
  - **m** can be a power of 2
- In the test files we provide additional input
  - **MaxStationsCount** the maximum number of stations contained in the testfile
  - Utilize it
- **Study the slide sections on Complexity Analysis of Separate Chaining Hashing**

Parties Array



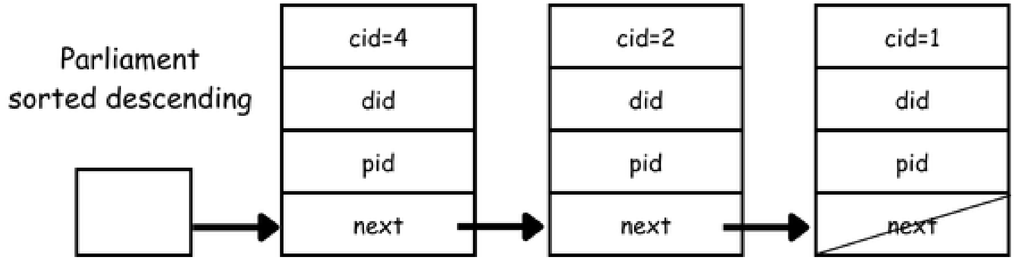
- **Parties**
  - array of 5 cells of type Party
- Type Party
  - **pid** unique party id
  - **electedCount** counter of elected candidates
  - **candidates** tree containing candidates registered to the Party

Candidates Binary Search Tree



- **Singly-Linked, Binary Search Tree Ordered by cid**
- contains elements of type Candidate
- **candidates** (member of Party)
  - pointer to the root node
- Type Candidate
  - **cid** unique candidate id
  - **did** district id the candidate is representing
  - **votes** counter of collected votes
  - **isElected** boolean indicating if candidate has been elected
  - **lc** pointer to left child node
  - **rc** pointer to right child node

Parliament List



- **Singly-Linked List, Ordered descending by cid**
- contains elements of type ElectedCandidate
- **Parliament**
  - pointer to the first node of the list (or NULL if list is empty)
- Type ElectedCandidate
  - **cid** unique candidate id
  - **did** district id the candidate is representing
  - **pid** party id the candidate is registered to
  - **next** pointer to next node

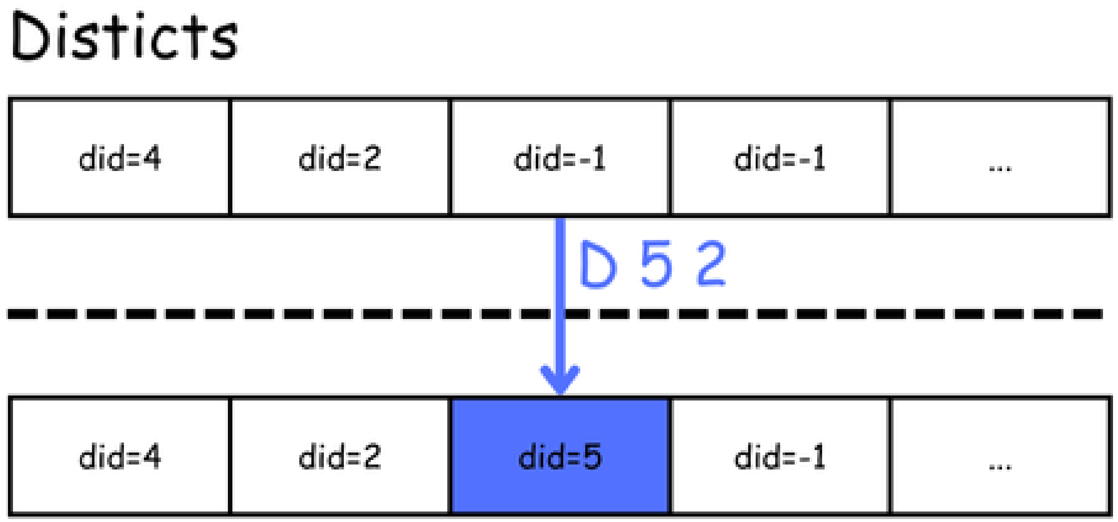
Events

A <MaxStationsCount> <MaxSid> Announce Elections

- Initialize Districts array
  - did = -1
  - blanks, invalids = 0
  - partyVotes = 0 in each cell
- Initialize Stations Hashtable
  - Pick hashtable capacity and assign it to a global - Utilize <MaxStationsCount>
  - Pick values essential to implementing universal hashing and assign them to globals - Utilize <MaxSid>
  - Stations = allocate memory for capacity number of chains
  - Initialize each chain as empty
- Initialize Parties array
  - pid = cell index
  - electedCount = 0
  - candidates = NULL
- Initialize Parliament = NULL

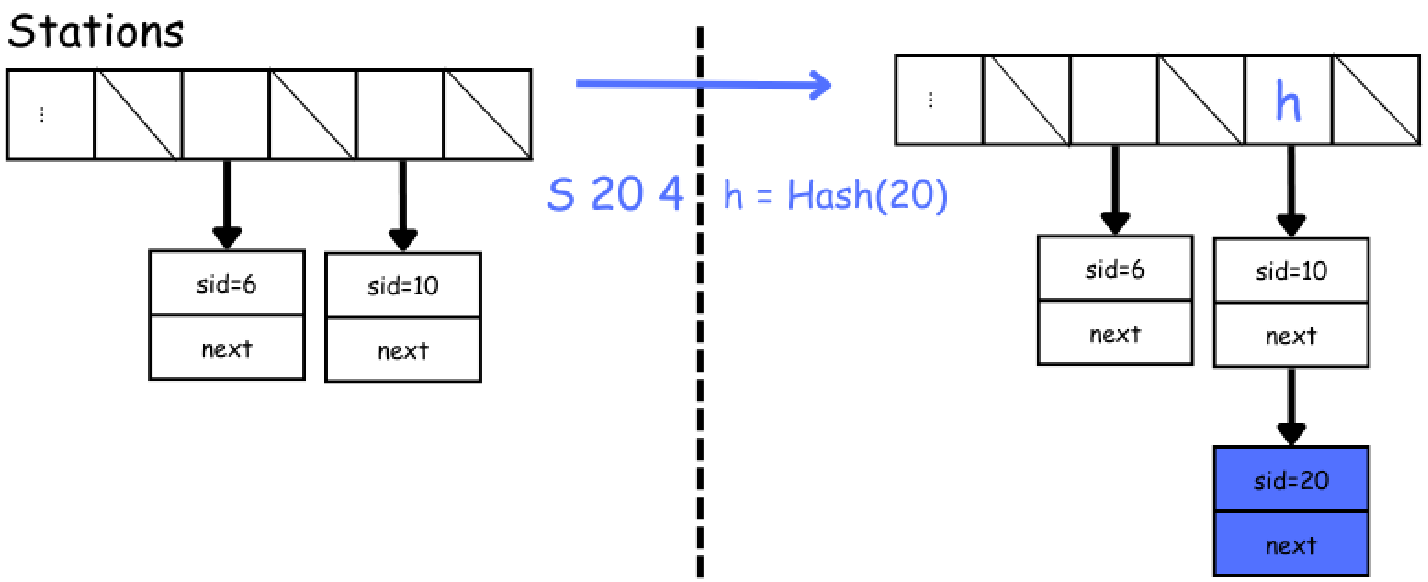
D <did> <seats>
Create District

- Find first empty slot (did == -1) of Districts array
  - O(log n) time
  - Recursive function, without globals
  - Similar to binary search
- Initialize it's fields
  - blanks, invalids, partyVotes = 0



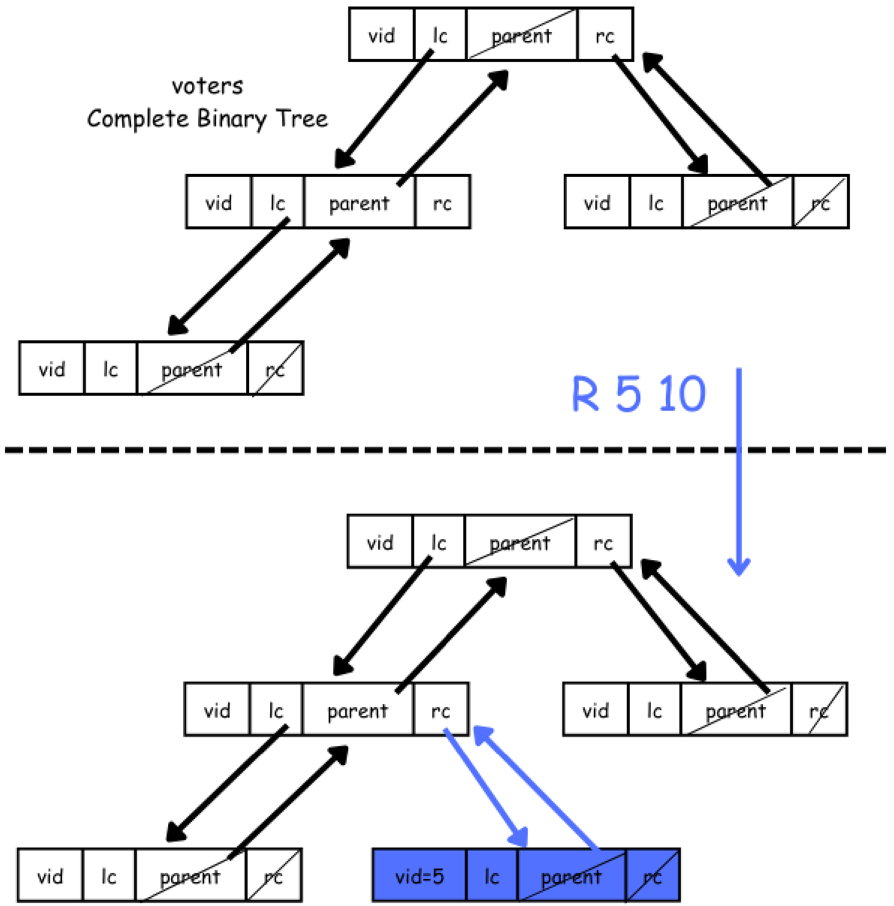
S <sid> <did>
Create Station

- Allocate memory for a Station
  - registered = 0
  - voters, next = NULL
- h = Hash(sid)
  - Hash function defined by you
- Insert to chain h
  - InsertSorted() by sid



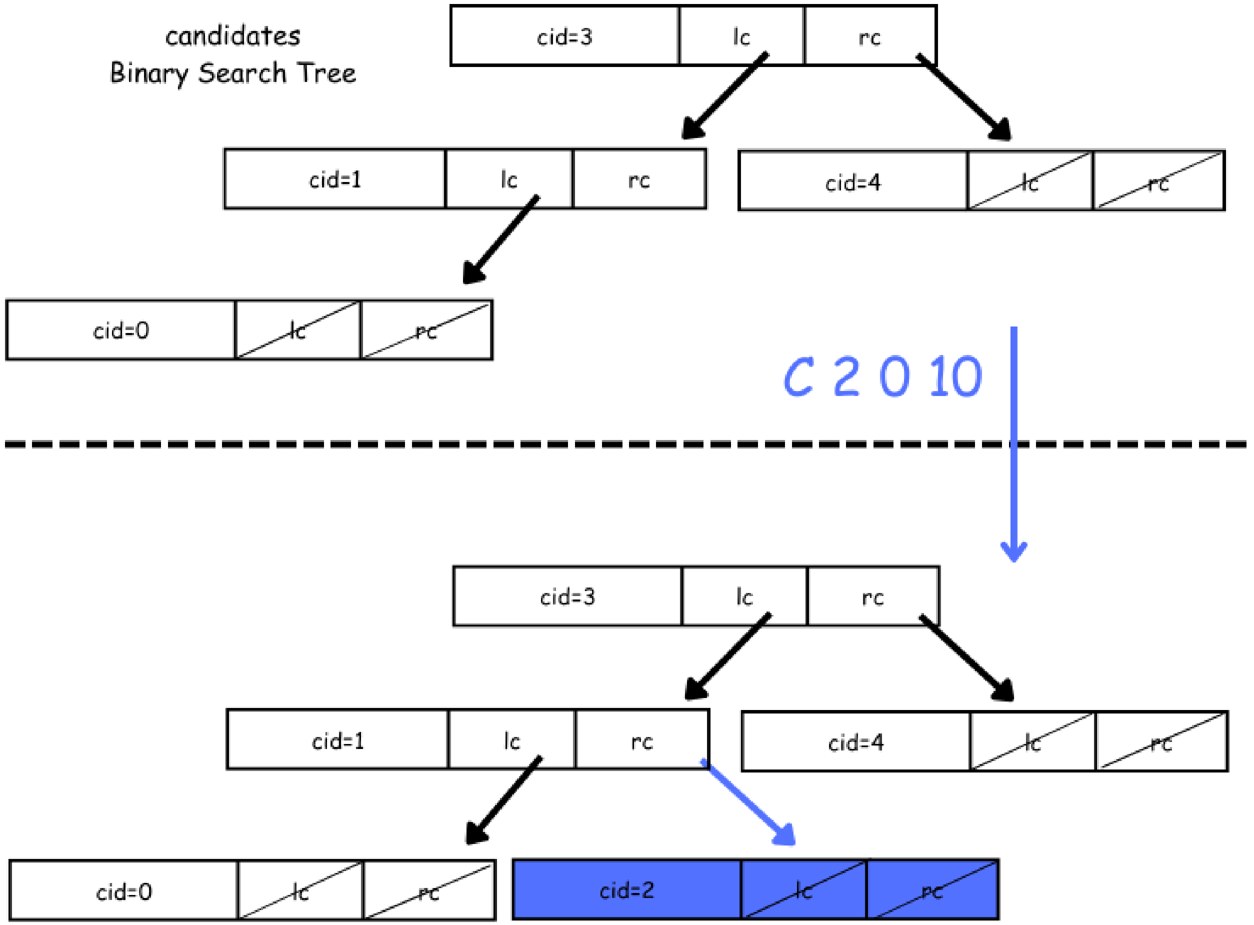
R <vid> <sid>
Register Voter

- Allocate memory for a Voter
  - voted ← false
  - parent, lc, rc ← NULL
- Find station <sid> in Stations Hashtable
  - Hashtable search
  - Amortized O(1)
- Insert to **voters** tree of Station <sid>
  - Complete Tree Insertion
  - Check exercises



C <cid> <pid> <did> Register Candidate

- Allocate memory for a Candidate
  - votes = 0
  - isElected = false
  - lc, rc = NULL
- Insert to candidates **BST** of Party <pid>
  - Binary Search Tree Insertion
  - O(BST Height)



V <vid> <sid> <cid> <pid> Vote

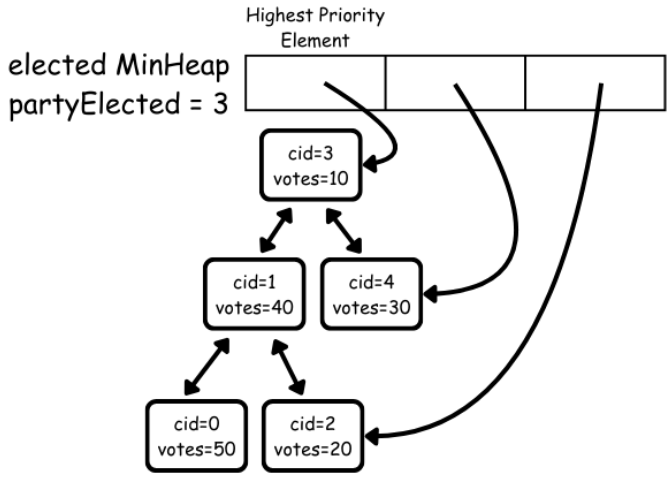
- Find s = station <sid> in Stations Hashtable
- Find v = voter <vid> in s
- v->voted = true
- Find d = district s->did in Districts array using simple linear search
- If <cid> == -1 then d->blanks += 1
- If <cid> == -2 then d->invalids += 1
- If <cid> >= 0
  - p = Party <pid>
  - Find c = candidate <cid> in p->candidates
    - BST Search, O(BST Height)
  - c->votes += 1
  - Find d1 = district c->did in Districts array using simple linear search
  - d1->partyVotes[<pid>] += 1

M <did>
Count Votes

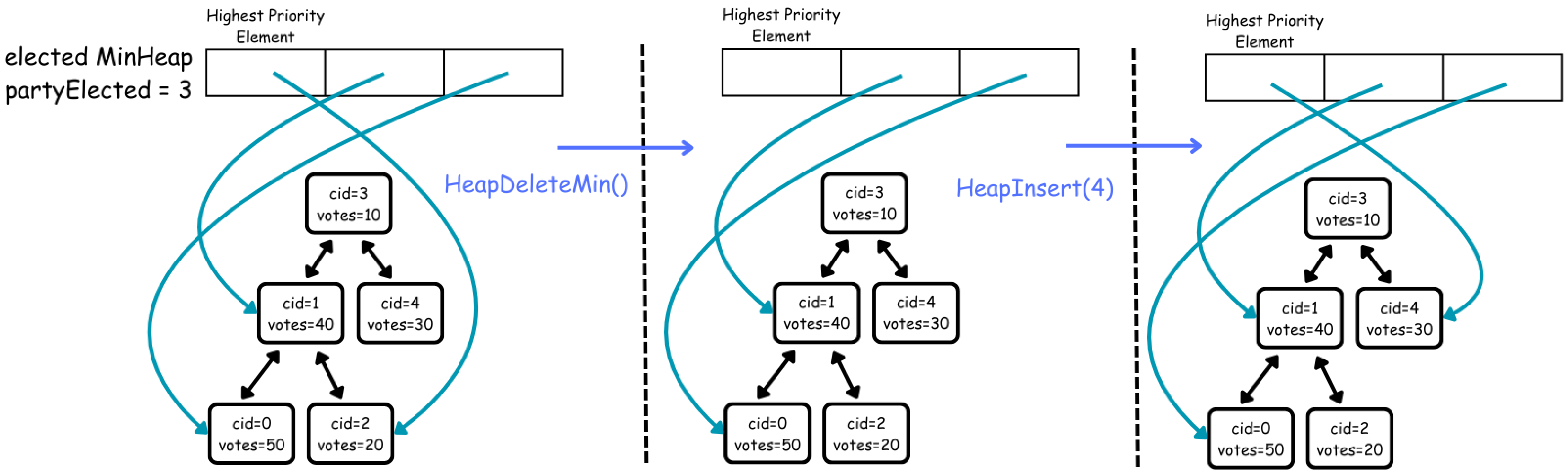
- Find d = district<did> in Districts array
- $\text{electoralQuota} = \dfrac{\text{sum(d->partyVotes)}}{\text{d->seats}}$
  - if d->seats == 0 then electoralQuota = 0
- For each p = Party <pid>
  - $\text{partyElected[<pid>]} = \dfrac{\text{d->partyVotes[<pid>]}}{\text{electoralQuota}}$
    - if electoralQuota == 0 then partyElected[<pid>] = 0
  - p.electedCount += partyElected[pid]
  - d->seats -= partyElected[<pid>]
- For each p = Party <pid>
  - **ElectPartyCandidatesInDistrict**(<pid>, <did>, partyElected[<pid>])

ElectPartyCandidatesInDistrict (partyElected, pid, did) → void

- Allocate Memory for array **elected**
  - holds pointers to Candidate
  - has capacity of partyElected elements
  - Implements MinHeap
  - On algorithm completion, holds pointers to elected Candidates
- Initialize elected with the first candidates in Party[pid].candidates where c.did == did
  - HeapInsert(c) each element
- For each c = candidate in Party[pid].candidates where c.did == did
  - **if (elected[0]->votes < c->votes)**
    - HeapDeleteMin()
    - HeapInsert(c)
- For each element in elected, set isElected = true



elected MinHeap example



N
Form Parliament

- Merge candidates of all parties, forming the Parliament List
    - Ignore non-elected candidates
    - Parliament List is sorted descending
    - Check exercises

BU <vid> <sid>
Bonus Unregister Voter

- Reverse of Register Voter
- Find station <sid> in Stations Hashtable
  - Hashtable search
  - Amortized O(1)
- Remove voter from member voters of Station <sid>
  - Complete Tree Removal

BF
Bonus Free Memory

- Free any memory you have allocated in your program
- Make sure you have no memory leaks
- Use the tool **valgrind** to catch leaks

Tips

- **Understand** your algorithm **before** coding it
- Test your algorithm with simple examples before coding
  - Use pen & paper, draw shapes
- After coding your algorithm test it with various inputs
  - Do this **before** moving the next algorithm
  - Make sure you are confident your code works before continuing
  - Use **gdb** to debug errors
- If an algorithm is too complex, split it to simpler parts
  - Apply the above for each part recursively
- Utilize **mailing list** and **office hours**