DIPARTIMENTO
DI INFORMATICA
SISTEMISTICA
E COMUNICAZIONE

Linguaggi di Programmazione AA 2022-2023
Introduzione a C e C++ (4)
Priority Queues

Marco Antoniotti
Gabriella Pasi
Fabio Sartori

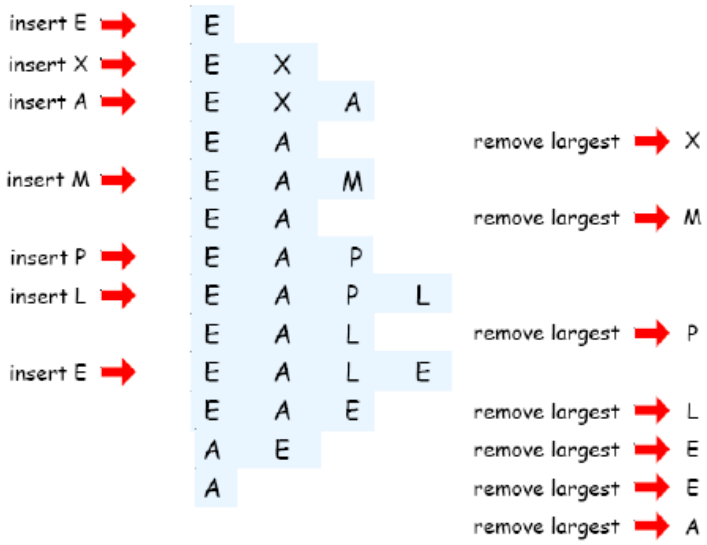
Code con priorità

- Unica assunzione
  - Gli elementi che sono parte dell'input sono confrontabili

- Operazioni fondamentali

Operazioni proprie dell' ADT

  - Inserimento
  - Rimozione del massimo (o del minimo)



Operazioni generiche su ADT

  - Copia
  - Concatenzione
  - Creazione
  - Distruzione
  - Controllo se vuoto

Code con priorità: applicazioni

- Applicazioni
  - Simulazione “event-driven” clienti in coda, sistemi di particelle
  - “Number crunching” riduzione di errori di arrotondamento
  - Compressione di dati codici di Huffman
  - Ricerca su grafi algoritmi di Dijkstra e Prim
  - Teoria dei numeri somma di potenze
  - Intelligenza artificiale algoritmo A*
  - Statistica manutenzione degli M valori più grandi
  - Sistemi operativi “load balancing”, gestione interruzioni
  - Ottimizzazione discreta “bin packing”, “scheduling”
  - Filtri spam filtri Bayesiani
- Generalizza
  - Pile, code, code randomizzate

Code con priorità: un esempio

- Problema:
  - Abbiamo un flusso di ***N*** dati in arrivo con ***N*** molto più grande della disponibilità di memoria del nostro calcolatore
  - Dobbiamo recuperare gli ***M*** elementi più grandi
  - Esempi
    - Rilevamento frodi: isolare le transazioni più grandi
    - Manutenzione file systems: trovare i files o le directories più grandi
- Soluzione: si usa una coda con priorità

Assumiamo di avere questa funzione

```
int main() {
      pq_priority_queue pq = pq_new_priority_queue(str_less, M + 1);
      char in[80];

      while (scanf("%s", in) != EOF) {
               pq_insert(pq, in);
               if (pq_count(pq) > M) pq_extract(pq);
      }
      while (! pq_is_empty(pq))
               printf("%s", (char*) pq_extract(pq);
      free(pq);
}
```

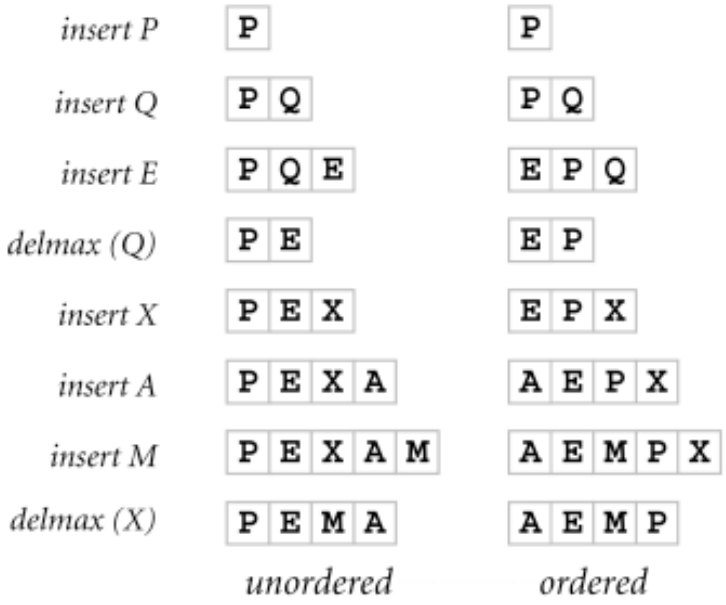
Code con priorità: implementazioni elementari

- Sfida: implementare entrambe le operazioni base in modo efficiente

- Implementazione elementare con un array ordinato o no

Implementazione	`insert`	`extract`
Array non ordinato	O(1)	O(N)
Array ordinato	O(N)	O(1)

Caso asintotico peggiore con N elementi



Implementazione con array non ordinato

- Una semplice implementazione

```
typedef int (* pq_compare) (void*, void*);
typedef struct _pq {
     void** pq;
     int size;
     int count;
     pq_compare compare;
} * pq_priority_queue;

pq_priority_queue
pq_new_priority_queue(pq_compare cf, int size);

void  pq_insert(pq_priority_queue, void*);
void* pq_extract(pq_priority_queue);
int   pq_count(pq_priority_queue);
bool  pq_is_empty(pq_priority_queue);
```

Implementazione con array non ordinato

```
pq_priority_queue
pq_new_priority_queue(pq_compare cf, int size) {

    pq_priority_queue pq
       = (pq_priority_queue) malloc(sizeof(struct _pq));

    /* Error checking here… */

    pq->pq = malloc(size * sizeof(void *));

    /* Error checking here… */

    pq->compare = cf;
    pq->count   = 0;
    pq->size    = size;
    return pq;
}
```

Implementazione con array non ordinato

```
void pq_insert(pq_priority_queue pq, void* item) {
     pq->pq[(pq->count)++] = item;
}

void pq_count(pq_priority_queue pq) {
     return pq->count;
}

bool pq_is_empty(pq_priority_queue pq) {
     return 0 == pq->count;
}
```

Implementazione con array non ordinato

```
void* pq_extract(pq_priority_queue pq) {
     int e = 0;
     int i;
     for (i = 0; i < pq->count; i++)
             if (pq->compare(pq->pq[i], pq->pq[e]))
                     e = i;
     exchange(pq->pq, e, pq->count - 1);
     return pq->pq[--(pq->count)];
}
```

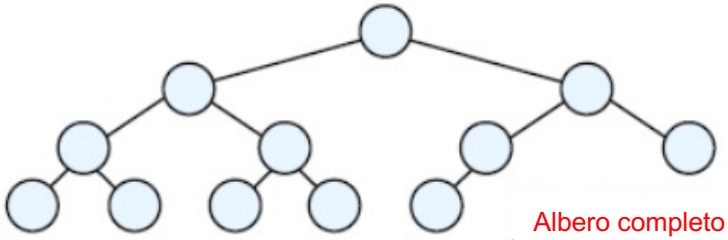
La complessità di questa operazione è lineare!

Possiamo far meglio?

La struttura dati "heap" (mucchio) binario

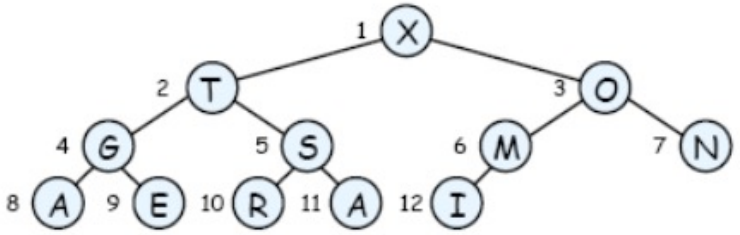
- Gli heaps ci permettono di implementare le operazioni **pq_extract** e **pq_insert** in tempo logaritmico
- Uno heap viene implementato (di solito) con un array in cui contenuto è interpretabile come un *albero binario completo* in "*heap-order*"

- Albero binario
  - Vuoto, oppure
  - Nodo con puntatori a due sottoalberi (destro e sinistro)



Albero completo bilanciato eccetto per l'ultimo livello

- Albero binario in heap order
  - Elementi nei nodi (chiavi)
  - Nessun elemento più grande (piccolo) degli elementi nei due sotto alberi



- Rappresentazione con un array
  - Si considerano i nodi per livello
  - Nessun puntatore è necessario dato che l'albero è completo

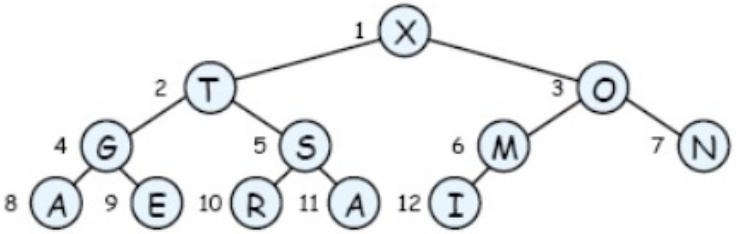
1	2	3	4	5	6	7	8	9	10	11	12
X	T	O	G	S	M	N	A	E	R	A	I

Proprietà degli heap (mucchi) binari

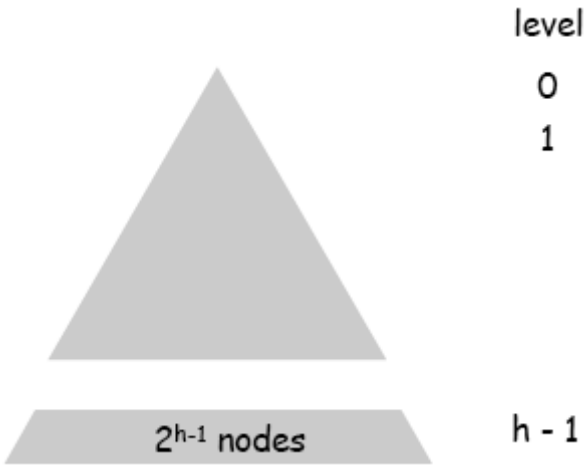
- Proprietà A: la chiave più piccola (grande) è in cima (alla radice dell'albero)
- Proprietà B: gli indici dell'array si possono usare per muoversi nell'albero
  - Nota: gli indici partono da 1
  - Il genitore del nodo in posizione K si trova in posizione K/2 (arrotondato)
  - I figli del nodo K si trovano in posizione 2K e 2K + 1
- Proprietà C: l'altezza di uno heap è

$$h = 1 + \lfloor \lg(N) \rfloor$$

  - Il livello *i* ha al più 2^i nodi
  - $1+2+4+\ldots+2^{h-1} \geq N$

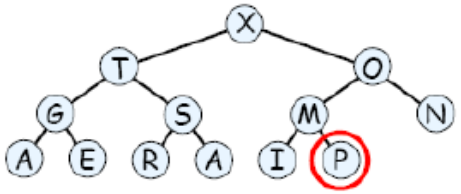


1	2	3	4	5	6	7	8	9	10	11	12
X	T	O	G	S	M	N	A	E	R	A	I

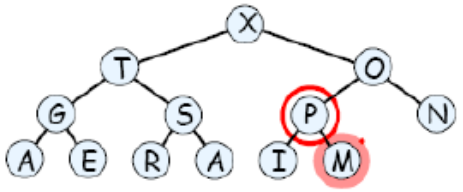


“Promozione” in uno heap

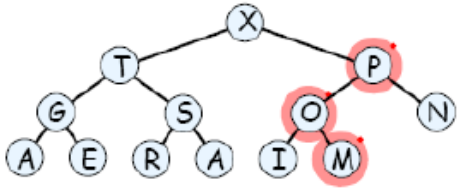
- Scenario: esattamente un nodo è più grande (piccolo) del suo genitore
  - Ovvero vengono violate le proprietà degli heaps



- Per eliminare la violazione
  - Si scambia il nodo con il genitore
  - Si ripete finchè le proprietà dello heap sono ripristinate




```
void swim(pq_priority_queue pq, int k) {
  while (k > 1
         && pq->compare(pq->pq[k/2], pq->pq[k])) {
    exchange(pq->pq, k, k/2);
    k = k/2;
  }
}
```

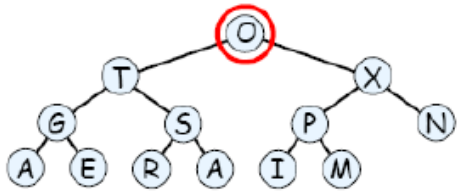


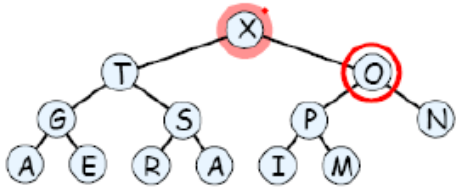
1	2	3	4	5	6	7	8	9	10	11	12	13
X	T	O	G	S	M	N	A	E	R	A	I	P
X	T	P	G	S	O	N	A	E	R	A	I	M

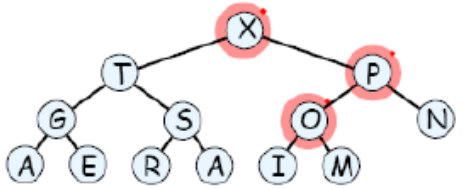
“Demozione” in uno heap

- Scenario: esattamente un nodo è più grande (piccolo) di un suo discendente
  - Per eliminare la violazione
    - Si scambia il nodo con il più grande (piccolo) dei suoi discendenti
    - Si ripete finchè le proprietà dello heap sono ripristinate

```
void sink(pq_priority_queue pq, int k) {
  while (2 * k <= pq->count) {
    int j = 2 * k;
    if (j < pq->count
          && pq->compare(pq->pq[j], pq->pq[j+1]))
       j++;
    if (! pq->compare(pq->pq[j], pq->pq[k]))
       break;
    exchange(pq->pq, k, j);
    k = j;
  }
}
```





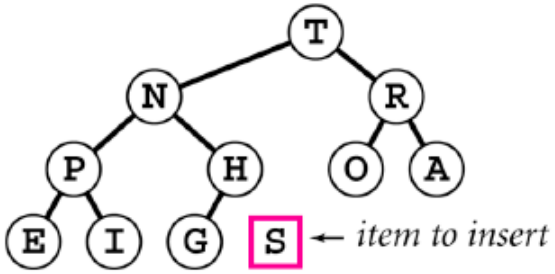


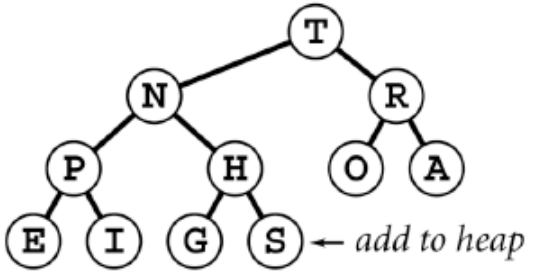
1	2	3	4	5	6	7	8	9	10	11	12	13
O	T	X	G	S	P	N	A	E	R	A	I	M
X	T	P	G	S	O	N	A	E	R	A	I	M

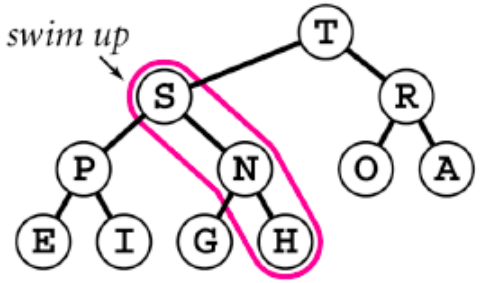
Inserimento in uno heap

- L'operazione di inserimento è molto semplice

```
void pq_insert(pq_priority_queue pq,
               void* item) {
     pq->pq[++pq->count] = item;
     swim(pq, pq->count);
}
```

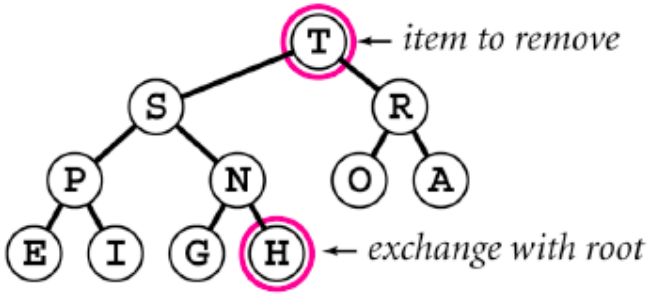




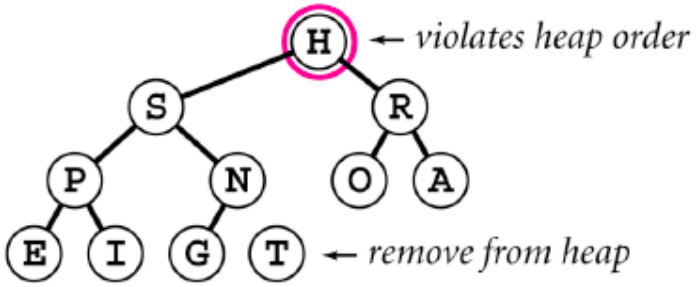


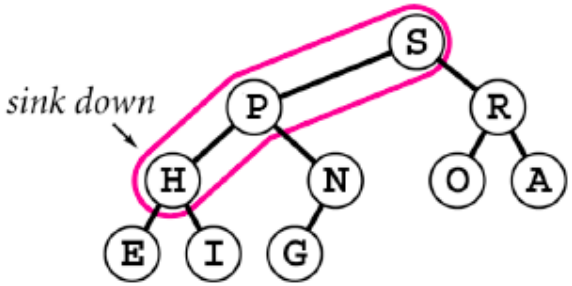
Estrazione

- Estrazione: si scambia la radice con l'ultimo elemento e poi si demuove la nuova radice




```
void* pq_extract(pq_priority_queue pq) {
  void* item = pq->pq[1];
  exchange(pq->pq, 1, pq->count--);
  sink(pq, 1);
  pq->pq[1 + pq->count] = NULL;
  return item;
}
```





Considerazioni ulteriori

- Ridimensionamento dell'array
  - Si può estendere l'array mediante il metodo "del raddoppio" quando la coda con priorità è piena
  - In questo modo ogni operazione richiede $O(\lg(N))$ tempo ammortizzato
- Chiavi immutabili
  - L'implementazione presentata assume che le chiavi non vengano mai modificate dal programma che utilizza la libreria
  - È bene che il programma cliente usi chiavi immutabili
- Altre operazioni (implementabili usando **sink** e **swim**)
  - Rimozione di un elemento arbitrario
  - Modifica di una chiave
    - Operazione fondamentale per alcune applicazioni

Sommario sulle prestazioni

- Caso peggiore per una coda con priorità di N elementi

Operazione	`insert`	`extract`	`top`
Array ordinato	O(N)	O(1)	O(1)
Lista ordinata	O(N)	O(1)	O(1)
Array disordinato	O(1)	O(N)	O(N)
Lista disordinata	O(1)	O(N)	O(N)
Heap	O(lg(N))	O(lg(N))	O(1)

Digressione: ordinamenti con heaps

- Primo passo: si costruisce uno heap con tutti gli elementi dell'array in input
  - Si usa uno heap ausiliario
  - Oppure, si ricostruisce uno heap "in-place"
- Secondo passo: ordinamento
  - Si "estraggono" gli elementi massimi (minimi) uno alla volta
  - Invece di annullare l'elemento posto in ultima posizione, lo si tiene così com'è
  - Alla fine, l'array dello heap sarà ordinato
- Numero di confronti richiesto: al più 2 N lg(N)

Importanza di Heapsort

- Heapsort è ottimo in tempo e spazio nel peggior caso
  - Tempo $O(N \lg(N))$
  - Spazio $O(N)$

- Mergesort richiede spazio $2N$
- Quicksort può richiedere $O(N^2)$ nel caso peggiore

- Ciononostante, heapsort
  - Ha un ciclo interno più complicato di quicksort
  - Su nuove architetture hardware può avere caratteristiche di "caching" non buone

- La STL C++ usa un algortimo di sorting ibrido tra quicksort, heapsort ed insertion-sort

DIPARTIMENTO
DI INFORMATICA
SISTEMISTICA
E COMUNICAZIONE

Applicazioni di code con priorità

Algoritmo A*

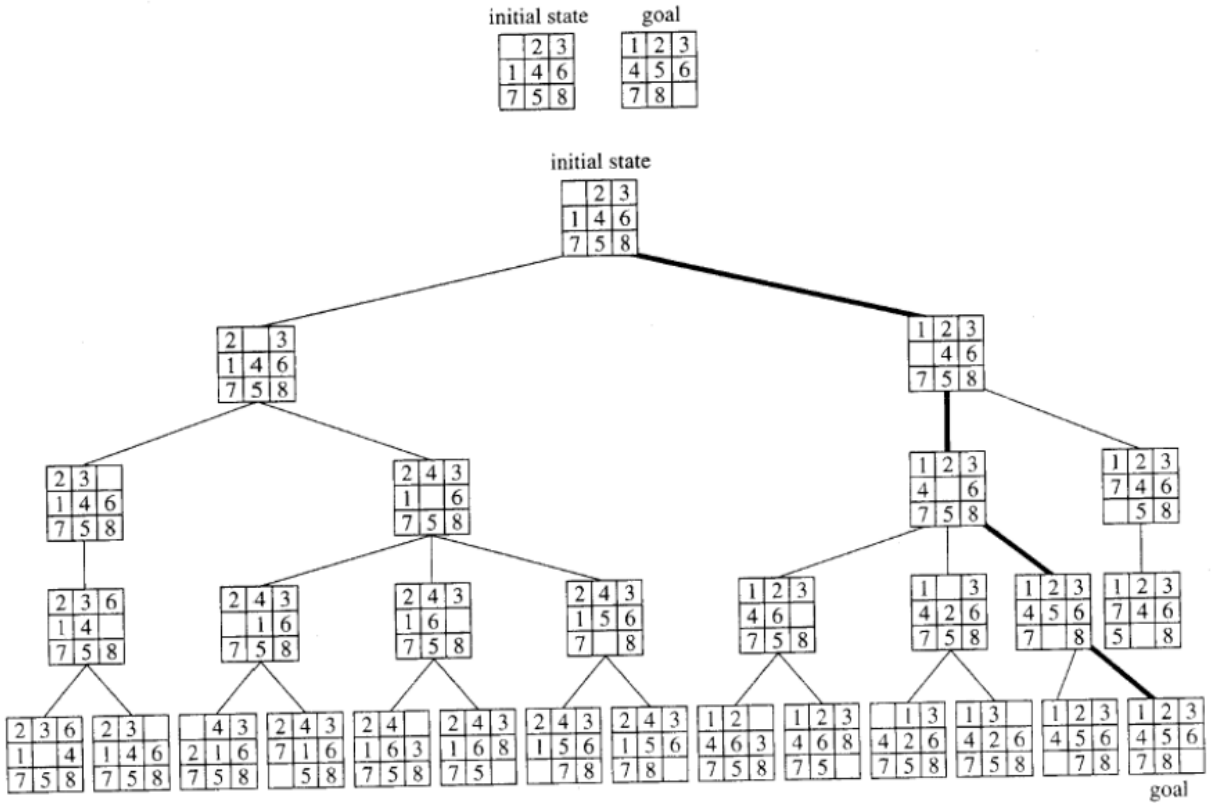
- Consideriamo il gioco del 15

http://www.javaonthebrain.com/java/puzz15/

Sam Loyd

Versione con 8 caselle

- Soluzione con una ricerca esaustiva (breadth first)

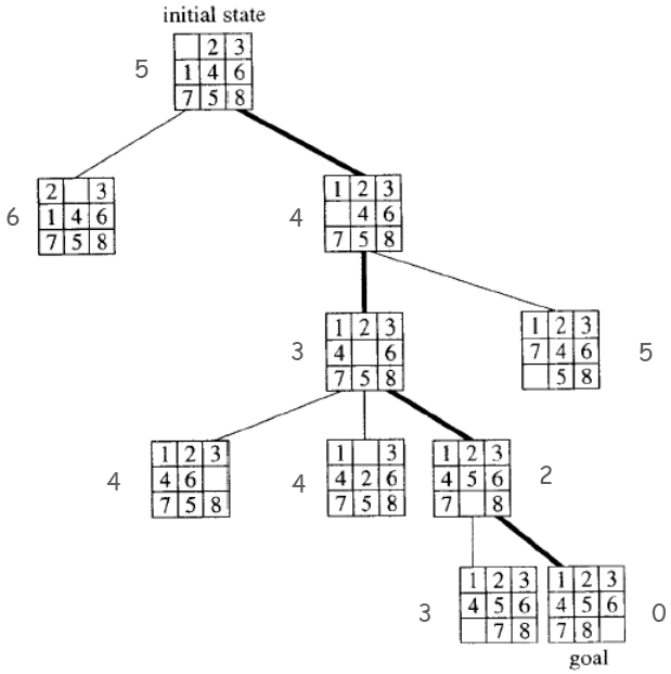


Pictures from *Sequential and Parallel Algorithms* by Berman and Paul.

Soluzione A* (gioco dell'8)

- Ricerca con priorità
  - Idea fondamentale: ricercare un cammino da uno stato iniziale ad uno finale in maniera più intelligente assegnando un punteggio ad ogni configurazione
  - Esempio 1: il punteggio è il numero di caselle fuori posto
  - Esempio 2: si usa come punteggio la "distanza di Manhattan" sommata alla "profondità" del nodo

- L'algoritmo A* può essere implementato con una coda a priorità

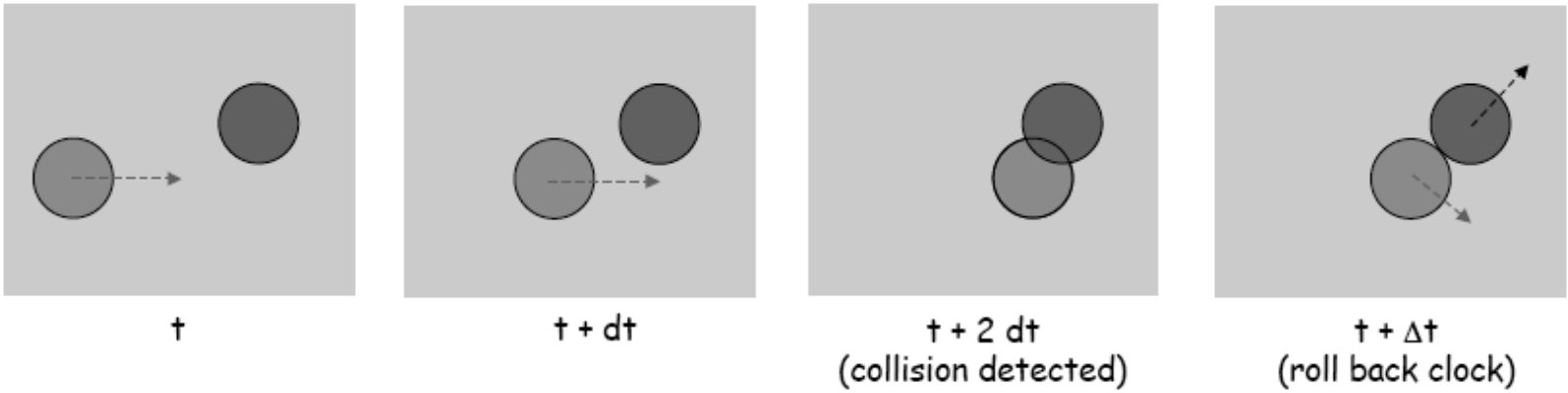


Simulazione “event-driven”

- Esempio: Simulazione della dinamica molecolare con sfere indeformabili
- Obiettivo: simulare N particelle che si muovono obbedendo alle leggi sugli urti elastici
- Modello con sfere indeformabili
  - Le particelle in movimento interagiscono tra di loro e con delle pareti tramite urti elastici
  - Ogni particella è una sfera di cui sono note posizione, velocità, massa e raggio
  - Non vi sono altre forze
- Significato: si mettono in relazione osservabili macroscopiche (temperatura, pressione, costante di diffusione) con dinamiche microscopiche (movimento individuale di atomi e molecole)
  - Maxwell e Boltzmann: si derivano le distribuzioni delle velocità delle molecole interagenti a partire dalla temperatura
  - Einstein: si spiega il moto Browniano dei pollini

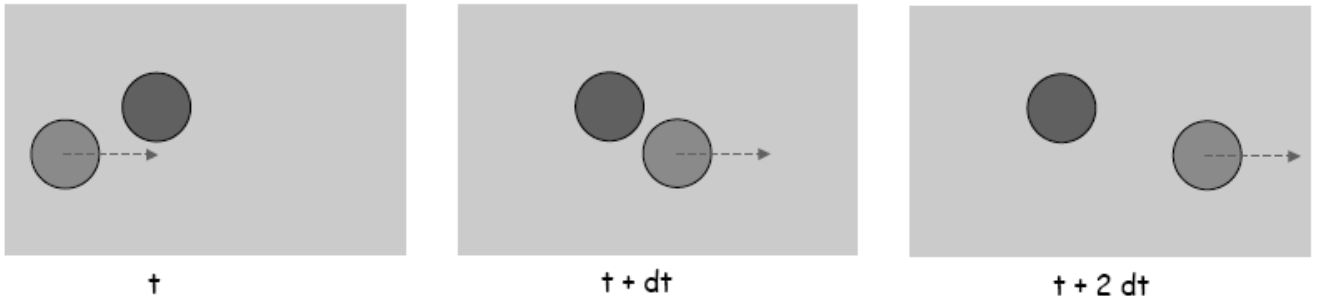
Soluzione “time-driven”

- Si discretizza il tempo in quanti di dimensione *dt*
- Si aggiornano le posizioni delle N particelle ad ogni *dt* e si controlla se vi sono sovrapposizioni (due centri di due particelle sono più vicini della somma dei loro raggi)
- Se vi sono sovrapposizioni si “torna indietro nel tempo” al momento della collisione, si ricomputano le velocità delle particelle e si continua la simulazione



Soluzione “time-driven”

- Problemi
  - N^2 controlli di sovrapposizione per quanto di tempo
  - È possibile perdere una sovrapposizione se il quanto di tempo è troppo grande
  - La simulazione rallenta molto se dt è molto piccolo



Soluzione “event-driven”

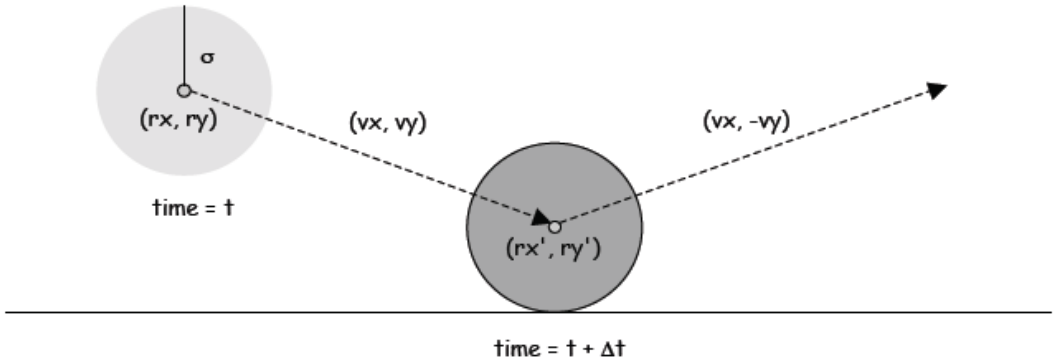
- Osservazioni
  - Tra un urto e l’altro le particelle si muovono in linea retta
  - I soli momenti “interessanti” sono quelli in cui si osserva un urto (particella-particella o particella-parete)
  - Possiamo mantenere una coda con priorità di questi eventi, con chiave il tempo
  - Estrarre il minimo elemento dalla coda con priorità equivale a trovare il “prossimo” urto
- Predizione degli urti
  - Data la velocità, la posizione ed il raggio di una particella, quando avverà un urto con un’altra particella o con la parete?
- Risoluzione di un urto
  - Se un urto accade, si aggiornano le velocità delle particelle coinvolte secondo le leggi regolanti gli urti elastici

Urto particella-parete

- Predizione urto
  - Particella di raggio r, posizione (rx, ry) che si muove con velocità (vx, vy)
  - Se e quando urterà una parete orizzontale?

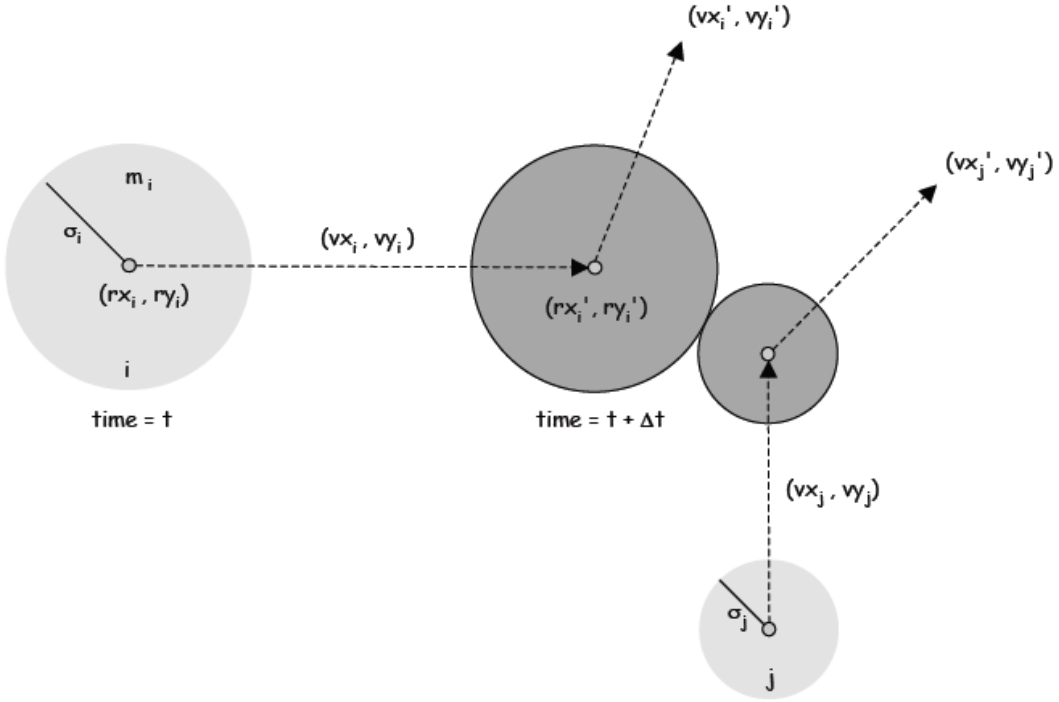
$$\Delta t = \begin{cases} \infty & \text{if } vy = 0 \\ (\sigma - ry)/vy & \text{if } vy < 0 \\ (1 - \sigma - ry)/vy & \text{if } vy > 0 \end{cases}$$

- Risoluzione urto: (vx', vy') = (vx, -vy)



Urto particella-particella

- Predizione urto
  - Particella i: raggio r_i, posizione (rx_i, ry_i), velocità (vx_i, vy_i)
  - Particella j: raggio r_j, posizione (rx_j, ry_j), velocità (vx_j, vy_j)
  - Se e quando la particella i urterà la particella j?



Urto particella-particella

- Predizione urto
  - Particella i: raggio r_i, posizione (rx_i, ry_i), velocità (vx_i, vy_i)
  - Particella j: raggio r_j, posizione (rx_j, ry_j), velocità (vx_j, vy_j)
  - Se e quando la particella i urterà la particella j?

$$\Delta t = \begin{cases} \infty & \text{if } \Delta v \cdot \Delta r \geq 0 \\ \infty & \text{if } d < 0 \\ - \dfrac{\Delta v \cdot \Delta r + \sqrt{d}}{\Delta v \cdot \Delta v} & \text{otherwise} \end{cases}$$

$$d = (\Delta v \cdot \Delta r)^2 - (\Delta v \cdot \Delta v)\,(\Delta r \cdot \Delta r - \sigma^2) \qquad \sigma = \sigma_i + \sigma_j$$

$$\Delta v = (\Delta vx,\ \Delta vy) = (vx_i - vx_j,\ vy_i - vy_j)$$
$$\Delta r = (\Delta rx,\ \Delta ry) = (rx_i - rx_j,\ ry_i - ry_j)$$

$$\Delta v \cdot \Delta v = (\Delta vx)^2 + (\Delta vy)^2$$
$$\Delta r \cdot \Delta r = (\Delta rx)^2 + (\Delta ry)^2$$
$$\Delta v \cdot \Delta r = (\Delta vx)(\Delta rx) + (\Delta vy)(\Delta ry)$$

Urto particella-particella

- Risoluzione urto
  - Quando due particelle si urtano, come cambiano le loro velocità?

$$
\left.
\begin{aligned}
vx_i' &= vx_i + Jx / m_i \\
vy_i' &= vy_i + Jy / m_i \\
vx_j' &= vx_j - Jx / m_j \\
vy_j' &= vx_j - Jy / m_j
\end{aligned}
\right\}
$$

Newton's second law (momentum form)

$$
Jx = \frac{J\,\Delta rx}{\sigma}, \quad Jy = \frac{J\,\Delta ry}{\sigma}, \quad J = \frac{2\,m_i\,m_j\,(\Delta v \cdot \Delta r)}{\sigma(m_i + m_j)}
$$

Impulso dovuto ad una forza normale (conservazione di energia, conservazione del momento)

Soluzione “event-driven”

- Inizializzazione
  - Si riempie la coda con priorità PQ con una insieme di tutti i possibili urti ($\approx N^2$) data la configurazione iniziale

- Ciclo principale
  - Si estrae l’evento incombente da PQ (particelle p1 e p2, tempo = t)
  - Se l’evento non è più “valido” lo si ignora
  - Si aggiornano le posizioni di tutte le particelle al tempo t, seguendo traiettorie lineari
  - Si aggiornano le velocità delle particelle p1 e p2 partecipanti all’urto
  - Si predicono i possibili futuri urti che coinvolgono le particelle p1 e p2 e li si inseriscono in PQ

Conclusioni

- Le code con priorità sono una struttura dati fondamentale
- Le loro applicazioni sono molteplici
  - Ordinamenti
  - Ricerche su grafo
  - Ottimizzazione
  - Simulazione event-driven
- Operazioni fondamentali
  - Inserimento O(lg(N))
  - Estrazione minimo (massimo) O(lg(N))
- Caratteristiche
  - Dati mantenuti in ordine parziale
- Altre operazioni
  - Modifica di chiave
  - Concatenazione di due heaps
- Variazioni (tutte le operazioni molto efficienti in tempo ammortizzato)
  - Heaps binomiali
  - Heaps di Fibonacci