7.4 Geneeriset algoritmit

Standardikirjasto tarjoaa useimmat tähän mennessä käsitellyistä algoritmeista.

Algoritmit on kaikki toteutettu funktiomalleina, jotka saavat kaikki tarvitsemansa tiedon käsiteltävistä säiliöistä parametrien avulla.

Algoritmeille ei kuitenkaan koskaan anneta parametrina kokonaisia säiliöitä vaan ainoastaan iteraattoreita niihin.

- algoritmeilla voidaan käsitellä myös säiliön osia kokonaisten säiliöiden sijasta
- algoritmi voi saada parametrinaan iteraattoreita erityyppisiin säiliöihin, jolloin yhdellä funktiokutsulla voidaan yhdistää esimerkiksi vectorin ja listan sisällöt ja tallettaa tulos joukkoon
- algoritmien toimintaa voidaan muuttaa iteraattorisovittimien avulla
- ohjelmoija voi toteuttaa omiin tietorakenteisiinsa iteraattorit, jonka jälkeen algoritmit toimivat myös niille

Kaikkia algoritmeja ei kuitenkaan pystytä suorittamaan kaikille tietorakenteille tehokkaasti.
$\Rightarrow$ osa algoritmeista hyväksyy parametreikseen vain tietyn iteraattorikategorian iteraattoreita.

- tämä takaa algoritmien tehokkuuden, koska kaikki iteraattorin tarjoamat operaatiot ovat vakioaikaisia
- jos iteraattori on väärää tyyppiä, annetaan käännösaikainen virhe-ilmoitus

  $\Rightarrow$ jos algoritmille annetaan tietorakenne, jolle sitä ei voida toteuttaa tehokkaasti, se ei edes käänny

Standardikirjaston algoritmit ovat kirjastossa `algorithm`. Lisäksi standardi määrittelee C-kielen algoritmikirjaston `cstdlib`.

jakaa algoritmit kolmee pääryhmään: muuttamattomat sarjalliset operaatiot, muuttavat sarjalliset operaatiot ja järjestäminen sekä siihen liittyvät operaatiot.

Seuraavaksi lyhyt kuvaus joistakin kurssin kannalta kiinnostavimmista algoritmeista (näiden lisäksi on vielä

runsaasti suoraviivaisia selaamiseen yms. perustuvia algoritmeja):

Puolitushaku

- `binary_search(`eka`,` $loppu$`,` $arvo$`)` kertoo onko $arvo$ järjestetyssä jononpätkässä
  - eka ja $loppu$ ovat iteraattoreita, jotka osoittavat etsittävän alueen alkuun ja loppuun, muttei välttämättä säiliön alkuun ja loppuun
- samaa arvoa voi olla monta peräkkäin

  $\Rightarrow$ `lower_bound` ja `upper_bound` palauttavat sen alueen rajat, jolla on $arvo$a
  - alaraja on, yläraja ei ole mukana alueessa
- rajat saa myös pariksi yhdistettynä yhdellä etsinnällä: `equal_range`
- vertaa BIN-SEARCH sivu 74

Järjestämisalgoritmit

- `sort`($alku$, $loppu$) ja `stable_sort`($alku$, $loppu$)
- `sort`in suoritusaika $O(nlogn)$ ja `stable_sort`in $O(nlogn)$ jos tarpeeksi lisämuistia on saatavilla, muuten $O(nlog^2n)$
- järjestelyalgoritmit vaativat parametreikseen hajasaanti-iteraattorit

  $\Rightarrow$ eivät toimi listoille, mutta niissä on oma `sort` (ja ei-kopioiva `merge`) jäsenfunktiona
- löytyy myös järjestäminen, joka lopettaa, kun halutun mittainen alkuosa on järjestyksessä: `partial_sort`($alku$, $keski$, $loppu$)
- lisäksi `is_sorted`($alku$, $loppu$) ja `is_sorted_until`($alku$, $loppu$)

`nth_element(` eka, $\ddot{a}nn\ddot{a}s$, $loppu$ `)`

- etsii alkion, joka järjestetyssä säiliössä olisi kohdalla $\ddot{a}nn\ddot{a}s$
- muistuttaa algoritmia RANDOMIZED-SELECT
- iteraattoreiden tulee olla hajasaanti-iteraattoreita

Ositus (partitiointi)

- `partition(`eka`,` $loppu$`,` $ehtofunktio$`)` epävakaa, erikseen `stable_partition`.
- `stable_partition(`eka`,` $loppu$`,` $ehtofunktio$`)` vakaa, mutta hitaampi ja/tai varaa enemmän muistia
- järjestää välillä eka - $loppu$ olevat alkiot siten, että ensin tulevat alkiot, joille $ehtofunktio$ palauttaa true ja sitten, ne joille se palauttaa false.
- vrt. QUICK-SORTn yhteudessä esitelty PARTITION
- partition on tehokkuudeltaan lineaarinen
- lisäksi `is_partitioned` ja `partition_point`

`merge(` $alku1$ `,` $loppu1$ `,` $alku2$ `,` $loppu2$ `,` $maali$`)`

- Algoritmi limittää välien $alku1$ - $loppu1$ ja $alku2$ - $loppu2$ alkiot ja kopioi ne suuruusjärjestyksessä iteraattorin $maali$ päähän
- algoritmi edellyttää, että alkiot yhdistettävillä väleillä ovat järjestyksessä
- vertaa sivun 45 MERGE
- algoritmi on lineaarinen
- $alku$- ja $loppu$-iteraattorit ovat syöttöiteraattoreita ja $maali$ on tulostusiteraattori

Keot

- STL:stä löytyy myös vastineet luvun 3.1 kekoalgoritmeille
- `push_heap(` eka `,` $loppu$`)` HEAP-INSERT
- `pop_heap(` eka `,` $loppu$ `)` vaihtaa huippualkion viimeiseksi (eli paikkaan $loppu - 1$) ja ajaa HEAPIFY:n osalle $eka \ldots loppu - 1$
  - vrt. HEAP-EXTRACT-MAX
- `make_heap(` eka `,` $loppu$`)` BUILD-HEAP

- `sort_heap(` eka , $loppu$ `)` HEAPSORT
- lisäksi `is_heap` ja `is_heap_until`
- iteraattoreiden tulee olla hajasaanti-iteraattoreita

Joukko-operaatiot

- C++:n standardikirjasto sisältää tätä tukevia funktioita
- `includes(` $eka1$, $loppu1$, $eka2$, $loppu2$ `)` osajoukko $\subseteq$
- `set_union(` $eka1$, $loppu1$, $eka2$, $loppu2$, $tulos$ `)` unioni $\cup$
- `set_intersection(...)` leikkaus $\cap$
- `set_difference(...)` erotus -
- `set_symmetric_difference(...)`
- $alku$- ja $loppu$-iteraattorit ovat syöttöiteraattoreita ja $tulos$ on tulostusiteraattori

`find_first_of(` $eka1$, $loppu1$, $eka2$, $loppu2$ `)`

- lopussa voi lisäksi olla tutkittavia alkioita rajaava ehto

- etsii ensimmäisestä jonosta ensimmäisen alkion, joka on myös toisessa jonossa
- jono voi olla taulukko, lista, joukko, . . .
  - yksinkertainen toteutus on hitaimmillaan $\Theta(nm)$, missä n ja m ovat jonojen pituudet
- toinen jono selataan jokaiselle ensimmäisen jonon alkiolle
  – hitain tapaus kun ei löydy
  $\Rightarrow$ hidasta, jos molemmat jonot pitkiä
- toteutus saataisiin yksinkertaiseksi, nopeaksi ja muistia säästäväksi vaatimalla, että jonot ovat järjestyksessä

HUOM! Mikään STL:n algoritmi ei automaattisesti tee säiliöihin lisäyksiä eikä poistoja, vaan ainoastaan muokkaa olemassa olevia alkioita.

- esimerkiksi `merge` ei toimi, jos sille annetaan tulostusiteraattoriksi iteraattori tyhjän säiliön alkuun
- jos tulostusiteraattorin halutaan tekevän lisäyksiä kopioinnin sijasta, tulee käyttää iteraattorisovitinta lisäysiteraattori

7.5 Lambdat: ()(){}

Algoritmikirjaston yhteydessä on paljon tilanteita, joissa on tarve välittää funktiolle toiminnallisuutta

- esim. `find_if`, `for_each`, `sort`

Lambdat ovat nimettömiä, määrittelemättömän tyyppisiä funktion kaltaisia. Ne ottavat parametreja, palauttavat paluuarvon ja pystyvät viittaamaan luontiympäristönstä muuttujiin sekä muuttamaan niitä.

Syntaksi: `[ympäristö](parametrit)->paluutyyppi {runko}`

- Jos lambda ei viittaa ympäristöönsä `ympäristö` on tyhjä
- `parametrit` voi puuttua
- jos `->paluutyyppi`ä ei ole annettu, se on void. Yksittäisestä return-lauseesta se voidaan päätellä
- esim. `[](int x, int y){ return x+y;}`

```
for_each( v.begin(), v.end(),[] (int val) {cout<<val<<endl;});
std::cin >> raja; //paikallinen muuttuja
std:find_if(v.begin(), v.end(),[raja](int a){return a<raja;});
```

STL:n algoritmeja voi ajatella nimettyinä erityissilmukoina, joiden runko lambda on

```
bool kaikki = true;
for (auto i : v)
{
    if (i%10 != 0) {
        kaikki = false;
        break;
    }
}
if (kaikki) {...}

if (std::all_of(v.begin(), v.end(), [](int i){return i%10==0;}){...}
```