7 C++:n standardikirjasto

Tässä luvussa käsitellään C++:n standardikirjaston tietorakenteita ja algoritmeja.

Tarkoituksena on käsitellä sellaisia asioita, joihin tulee kiinnittää huomiota, jotta kirjastoa tulisi käyttäneeksi tarkoituksenmukaisesti ja tehokkaasti.

7.1 Yleistä C++:n standardikirjastosta

Standardikirjasto standardoitiin C++-kielen mukana syksyllä 1998, ja sitä on jonkin verran laajennettu myöhemmissä versioissa. Uusin standardiversio on C++17 vuodelta 2017.

Kirjasto sisältää tärkeimmät perustietorakenteet ja algoritmit.

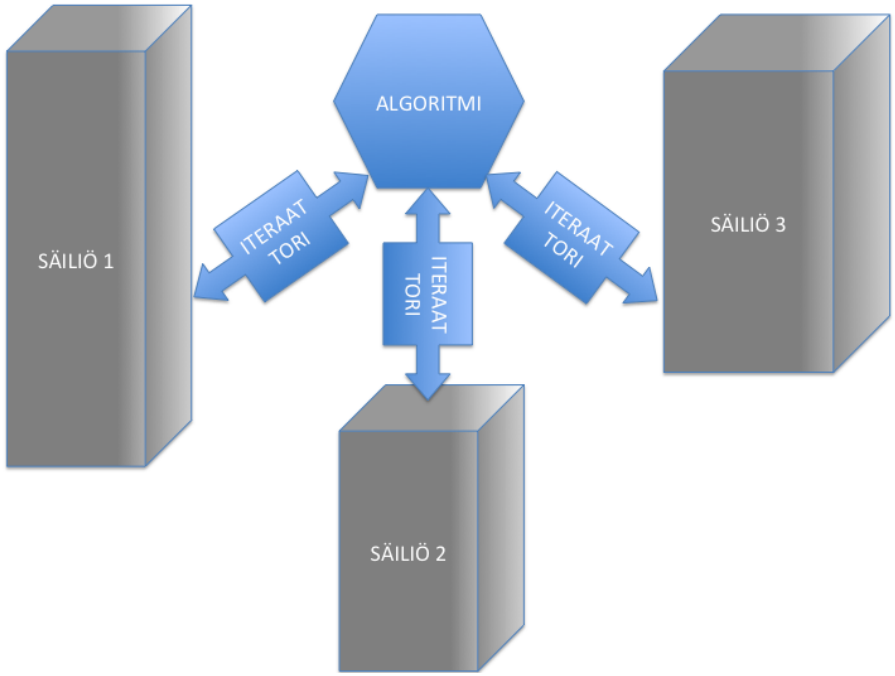
- useimmat tämän aineiston alkupuolen tietorakenteet ja algoritmit mukana muodossa tai toisessa

Rajapinnat ovat harkittuja, joustavia, geneerisiä ja tyyppiturvallisia.

Rajapintojen tarjoamien operaatioiden tehokkuudet on ilmaistu $\mathcal{O}$-merkinnällä.

Kirjaston geneerisyys on toteutettu käännösaikaisella mekanismilla: C++:n *malli* (template)

Tietorakennekurssin kannalta kiinnostavin standardikirjaston elementti on ns. STL (Standard Template Library): *säiliöt* eli kirjaston tarjoamat tietorakenteet, *geneeriset algoritmit* sekä *iteraattorit*, joiden avulla säiliöiden alkioita käsitellään.



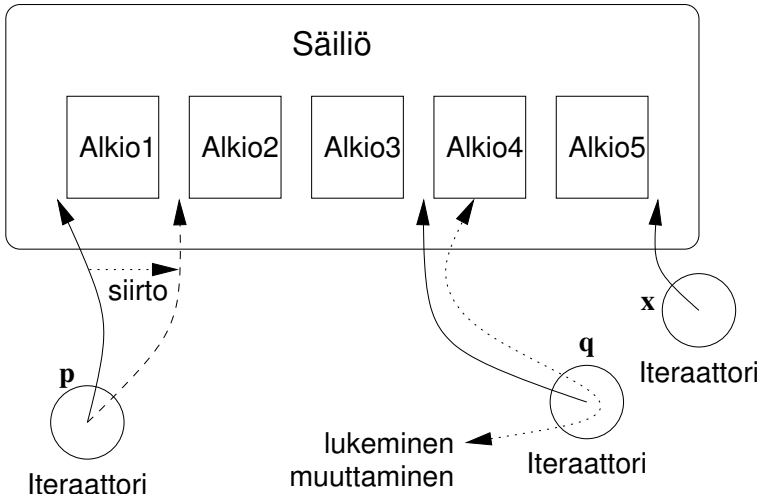
Kuva 12: STL:n osaset

C++11:n mukana tulleet *lambdat* ovat myös keskeisiä

7.2 Iteraattorit

Kaikki standardikirjaston tietorakenteet näyttäytyvät meille varsin samanlaisina mustina laatikoina, joista oikeastaan tiedämme ainoastaan, että ne sisältävät tallettamiamme alkioita ja että ne toteuttavat tietyt rajapintafunktiot.

Pystymme käsittelemään säiliöiden sisältöä ainoastaan noiden rajapintafunktioiden sekä iteraattorien avulla.



Iteraattorit ovat kahvoja tai "kirjanmerkkejä" tietorakenteen alkioihin.

- kukin iteraattori osoittaa joko tietorakenteen alkuun, loppuun tai kahden alkion väliin.
- säiliöiden rajapinnassa on yleensä funktiot **begin()** ja **end()**, jotka palauttavat säiliön alkuun ja loppuun osoittavat iteraattorit
- iteraattorin läpi pääsee käsiksi sen oikealla puolella olevaan alkioon, paitsi jos kysymyksessä on *käänteisiteraattori* (reverse iterator), joilloin sen läpi käsitellään vasemmanpuoleista alkiota
- käänteisiteraattorille myös siirto-operaatiot toimivan käänteisesti, esimerkiksi ++ siirtää iteraattoria pykälän vasemmalle
- **begin()**:ä ja **end()**:ä vastaavat käänteisiteraattorit saa **rbegin()**:llä ja **rend()**:llä

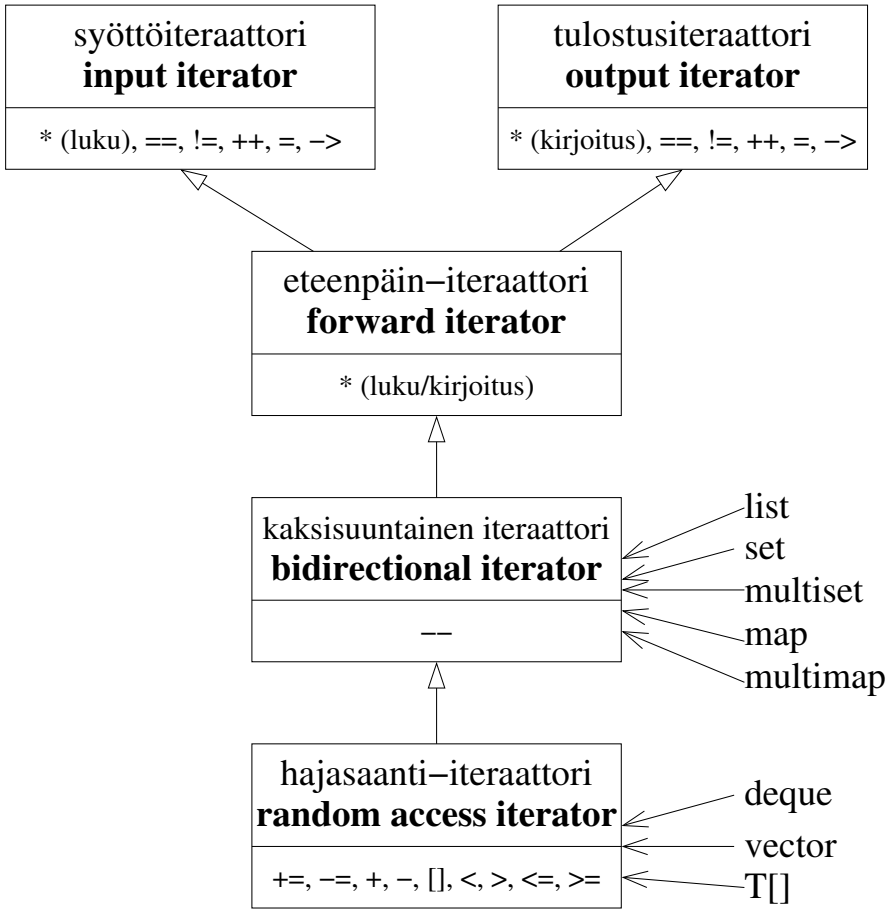
- nimensä mukaisesti iteraattoria voi siirtää säiliön sisällä, ja sen avulla säiliön voi käydä läpi
- iteraattorin avulla voi lukea ja kirjoittaa
- säiliöön lisättävien ja siitä poistettavien alkioiden sijainti yleensä ilmaistaan iteraattoreiden avulla

Kullakin säiliöllä on oma iteraattorityyppinsä.

- eri säiliöt tarjoavat erilaisia mahdollisuuksia siirtää iteraattoria nopeasti paikasta toiseen (vrt. taulukon/listan mielivaltaisen alkion lukeminen)
- suunnitteluperiaatteena on ollut, että kaikkien iteraattoreille tehtävien operaatioiden tulee onnistua vakioajassa, jotta geneeriset algoritmit toimisivat luvatulla tehokkuudella riippumatta siitä mikä iteraattori niille annetaan
- iteraattorit voidaan jakaa kategorioihin sen mukaan, millaisia vakioaikaisia operaatioita ne pystyvät tarjoamaan

Syöttöiteraattorin (input iterator) avulla voi vain lukea alkioita, mutta ei muuttaa niitä

- iteraattorin osoittaman alkion arvon voi lukea (*p)
- iteraattorin osoittaman alkion kentän voi lukea tai kutsua sen jäsenfunktiota (p->)
- iteraattoria voi siirtää askeleen eteenpäin (++p tai p++)
- iteraattoreita voi sijoittaa ja vertailla toisiinsa (p=q, p==q, p!=)



Tulostusiteraattori (output iterator) on kuten syöttöiteraattori, mutta sen avulla voi vain muuttaa alkioita. (*p=x)

Eteenpäin-iteraattori (forward iterator) on yhdistelmä syöttö- ja tulostusiteraattorien rajapinnoista.

Kaksisuuntainen iteraattori (bidirectional iterator) osaa lisäksi siirtyä yhden askeleen kerrallaan taaksepäin. (--p tai p--)

Hajasaanti-iteraattori (random access iterator) on kuin kaksisuuntainen iteraattori, mutta sitä voi lisäksi siirtää mielivaltaisen määrän eteen- tai taaksepäin.

- iteraattoria voi siirtää n askelta eteen- tai taaksepäin (p+=n, p-=n, q=p+n, q=p-n)
- iteraattorista n:n alkion päässä olevan alkion pystyy lukemaan ja sitä pystyy muokkaamaan (p(n))
- kahden iteraattorin välisen etäisyyden pystyy selvittämään (p-q)
- iteraattoreiden keskinäistä suuruusjärjestystä voi vertailla, iteraattori on toista “pienempi”, jos sen paikka on säiliössä ennen tätä (p<q, p<=q, p>q, p>=q)

Iteraattorit ovat osoitinabstraktio $\rightarrow$ Iteraattorien operaatioiden syntaksi muistuttaa selvästi C++:n osoitinsyntaksia.

Iteraattorit saadaan otettua käyttöön komennolla
`#include <iterator>`

Oikean tyyppinen iteraattori saadaan luotua esimerkiksi seuraavalla syntaksilla.

```
säiliö<talletettava tyyppi>::iterator p;
```

Iteraattoreiden kanssa avainsana `auto` on käyttökelpoinen:

```
auto p = begin( säiliö );
// → std::vector<std::string>::iterator
```

Säiliöihin tehtävät poistot ja lisäykset saattavat *mitätöidä* säiliöön jo osoittavia iteraattoreita.

- ominaisuus on säiliö-kohtainen, joten sen yksityiskohdat käsitellään säiliöiden yhteydessä

Tavallisten iteraattorien lisäksi STL tarjoaa joukon iteraattorisovittimia.

- niiden avulla voidaan muunnella geneeristen algoritmien toiminnallisuutta
- edellä mainitut *käänteisiteraattorit* (reverse iterator) ovat iteraattorisovittimia
- *lisäysiteraattorit* (insert iterator/inserter) ovat tärkeitä iteraattorisovittimia.
  - ne ovat tulostusiteraattoreita, jotka lisäävät alkioita halutulle paikalle säiliöön kopioimisen sijasta
  - säiliön alkuun lisäävän iteraattorin saa funktiokutsulla `front_inserter`(säiliö)
  - säiliön loppuun lisäävän iteraattorin saa funktiokutsulla `back_inserter`(säiliö)
  - annetun iteraattorin kohdalle lisäävän iteraattorin saa funktiokutsulla `inserter`(säiliö, paikka)

- *virtaiteraattorit* (stream iterator) ovat syöttö- ja tulostusiteraattoreita, jotka käyttävät säiliöiden sijaista C++:n tiedostovirtoja
  - `cin` virrasta haluttua tyyppiä lukevan syöttöiteraattorin saa syntaksilla `istream_iterator`<tyyppi> (`cin`)
  - `cout` virtaan haluttua tyyppiä pilkuilla erotettuna tulostavan tulostusiteraattorin saa syntaksilla `ostream_iterator`<tyyppi> (`cout, ','`)
- *siirtoiteraattorit* (move iterator) muuttavat alkion kopioinnin iteraattorin avulla alkion siirtämiseksi.

7.3 Säiliöt

Standardikirjaston säiliöt kuuluvat pääsääntöisesti kahteen kategoriaan rajapintojensa puolesta:

- *sarjat* (sequence)
  - alkioita pystyy hakemaan niiden järjestysnumeron perusteella
  - alkioita pystyy lisäämään ja poistamaan halutusta kohdasta
  - alkioita pystyy selaamaan järjestyksessä
- *assosiatiiviset säiliöt* (associative container)
  - alkiot sijoitetaan säiliöön *avaimen* määräämään kohtaan
  - talletettavien alkioiden avainten arvoja pitää pystyä vertaamaan toisiinsa oletusarvoisesti operaattorilla < järjestetyissä säiliöissä
- Rajapintojen tarjoamista jäsenfunktiosta näkee, mikä säiliön mielekäs käyttötarkoitus on

Kirjaston säiliöt:

Säiliötyyppi	Kirjasto

Sarjat	`array`
	`vector`
	`deque`
	`list`
	(`forward_list`)

Assosiatiiviset	`map` `set`
Järjestämättömät assosiatiiviset	`unordered_map` `unordered_set`
Säiliösovittimet	`queue` `stack`

Säiliöt noudattavat arvon välitystä.

- säiliö ottaa talletettavasta datasta kopion
- säiliö palauttaa kopioita sisältämästään tiedosta

  $\Rightarrow$ säiliön ulkopuolella tehtävät muutokset eivät vaikuta säiliön sisältämään dataan
- kaikilla säiliöihin talletettavilla alkioilla tulee olla kopiorakentaja ja sijoitusoperaattori.
  - perustyypeillä sellaiset ovat valmiina
- itse määriteltyä tyyppiä olevat alkiot kannattaa tallettaa osoittimen päähän
  - näinhän kannattaisi tehdä tehokkuussyistä joka tapauksessa

- C++11 tarjoaa kätevän työkalun muistinhallinnan helpottamiseksi `shared_pointer` tilanteissa, joissa useampi taho tarvitsee resurssia
  - sisältää sisäänrakennetun viitelaskurin ja tuhoaa alkion kun viitelaskuri nollautuu
  - deleteä ei tarvitse eikä saakaan kutsua
  - luominen: `auto pi = std::make_shared<Olio>(params);`
  - näppärä erityisesti jos halutaan tehdä kahden avaimen mukaan järjestetty tietorakenne:
    * sijoitetaan oheisdata `shared_pointer`in päähän
    * sijoitetaan osoittimet kahteen eri säiliöön kahden eri avaimen mukaan

Sarjat:

Taulukko `array<tyyppi>` on vakiokokoinen taulukko.

- Luodaan `std::array<tyyppi, koko> a = {arvo, arvo,...};`
- Indeksointi jäsenfunktiolla `.at()` tai `[]`-operaatiolla. Funktioilla `front()` ja `back()` voidaan käsitellä ensimmäistä ja viimeistä alkiota.
- Tarjoaa iteraattorit ja käänteisiteraattorit
- `empty()`, `size()` ja `max_size()`
- Funktion `data()` avulla päästään suoraan käsiksi sisällä olevaan taulukkoon

Taulukon operaatiot ovat vakioaikaisia, mutta `fill()` ja `swap()` ovat $O(n)$

Vektori `vector<tyyppi>` on lopustaan joustavarajainen taulukko

- Luodaan `vector<int> v {arvo, arvo, ...};`
- Vakioaikainen indeksointi `.at()`, `[]` sekä (tasatusti) vakioaikainen lisäys `push_back()` ja poisto `pop_back()` vektorin lopussa
- alkion lisääminen muualle `insert()`:illä ja poistaminen `erase`:lla on lineaarista, $O(n)$
- `emplace_back(args);` rakentaa alkion suoraan vektoriin
- Vektorin kokoa voi kasvattaa funktiolla `.resize(koko, alkuarvo);`
  - alkuarvo on vapaaehtoinen
  - tarvittaessa vektori varaa lisää muistia automaattisesti
  - muistia voi varata myös ennakkoon: `.reserve(koko)`, `.capacity()`
- iteraattorien mitätöitymistä tapahtuu seuraavissa tilanteissa
  - mikäli vectorille ei ole etukäteen varattu riittävää tilaa, voi mikä tahansa lisäys aiheuttaa kaikkien iteraattoreiden mitätöitymisen

  - poistot aiheuttavat mitätöitymisen ainoastaan poistopaikan jälkeen tuleville iteraattoreille
  - lisäykset keskelle aiheuttavat aina lisäyspaikan jälkeen tulevien iteraattoreiden mitätöitymisen
- `vector<bool>`:ille on määritelty erikoistoteutus, joka poikkeaa siitä mitä yleinen toteutus tekisi muistinkäytön tehostamiseksi
  - tavoite: mahdollistaa 1 bitti / alkio, kun tavallinen toteutus luultavasti veisi 1 tavu / alkio eli 8 bittiä / alkio

Pakka `deque<tyyppi>` on molemmista päistään avoin taulukko

- luodaan `deque<tyyppi> d {arvo, arvo, arvo...};`
- Rajapinta vektorin kanssa yhtenevä, mutta tarjoaa tehokkaan ($O(1)$ tasattu suoritusaika) lisäyksen ja poiston *molemmissa* päissä: `.push_front(alkio)`, `.emplace_front(args)`, `.pop_front()`
- iteraattorien mitätöitymistä tapahtuu seuraavissa tilanteissa
  - kaikki lisäykset voivat mitätöidä iteraattorit
  - poistot keskelle mitätöivät kaikki iteraattorit
  - kaikki paitsi päihin kohdistuvat lisäys- ja poisto-operaatiot voivat mitätöidä viitteet ja osoittimet

Lista on säiliö, joka tukee kaksisuuntaista iterointia

- luodaan `list<tyyppi> l {arvo, arvo, arvo };`
- lisäys ja poisto kaikkialla vakioaikaista, indeksointioperaatiota ei ole
- lisäys ja poisto eivät mitätöi iteraattoreita ja viitteitä (paitsi tietysti poistettuihin alkioihin)
- listalla on monenlaisia erikoispalveluja
  - `.splice(paikka, toinen_lista)` siirtää toisen listan nykyisen osaksi paikan eteen, $O(1)$.
  - `.splice(paikka, lista, alkio)` siirtää alkion toisesta tai samasta listasta paikan eteen, $O(1)$.
  - `.splice(paikka, lista, alku, loppu)` siirtää välin alkiot paikan eteen, $O(1)$ tai lineaarinen
  - `.merge(toinen_lista)` ja `.sort()`, vakaa, keskimäärin $O(nlogn)$
  - `.reverse()`, lineaarinen

Assosiatiiviset säiliöt:

Joukko `set<tyyppi>` ja **monijoukko** `multiset<tyyppi>` on dynaaminen joukko, josta voi

- etsiä, lisätä ja poistaa logaritmisessa ajassa
- selata suuruusjärjestyksessä tasatussa vakioajassa siten että läpikäynti alusta loppuun on aina lineaarinen operaatio
- alkioilla on oltava suuruusjärjestys "<"
- voi määritellä erikseen joko osana tyyppiä tai rakentajan parametrina
- tutkii yhtäsuuruuden kaavalla $\neg(x<y \vee y<x) \Rightarrow$ "<" määriteltävä järkevästi ja tehokkaaksi

Monijoukossa sama alkio voi olla moneen kertaan, joukossa ei

Luodaan `std::set<tyyppi> s {arvo, arvo, arvo...};`

- alkion arvon muuttaminen on pyritty estämään
  - sen sijaan pitää poistaa vanha alkio ja lisätä uusi
- mielenkiintoisia operaatioita:
  - `.find(alkio)` etsii alkion (monijoukolle ensimmäisen monesta), tai palauttaa `.end()` jollei löydä
  - `.lower_bound(alkio)` etsii ensimmäisen, joka on $\geq$ alkio
  - `.upper_bound(alkio)` etsii ensimmäisen, joka on $>$ alkio
  - `.equal_range(alkio)` palauttaa `make_pair( .lower_bound(alkio), .upper_bound(alkio) )`, mutta selviää yhdellä etsinnällä (joukolle ko. välin leveys on 0 tai 1)
  - joukoille `insert` palauttaa parin (paikka, lisättiin), koska alkiota ei saa lisätä, jos se on jo joukossa
- standardi lupaa, että iteraattorit eivät vanhene lisäyksessä ja poistossa (paitsi tietysti poistettuihin alkioihin kohdistuneet)

Kuvaus `map<avaimen_tyyppi, alkion_tyyppi>` ja **monikuvaus** `multimap<avaimen_tyyppi, alkion_tyyppi>`

- alkiot avain-oheisdatapareja
  - parin tyyppi on `pair<tyyppi1, tyyppi2>`
  - parin voi tehdä funktiolla `make_pair`:illä
  - parin kentät saa operaatioilla `.first()`, `.second()`
- luodaan

  ```
  std::map m<avain_tyyppi, alkio_tyyppi> m {{avain1, arvo1},
   {avain2, arvo2}, avain3, arvo3},...};
  ```

  esim.

  ```
  std::map<std::string,int> anim { {"bear",4}, {"giraffe",2},
   {"tiger",7} };
  ```

- `map`:ia voi poikkeuksellisesti indeksoida avaimen avulla $O(logn)$
  - Jos avainta ei löydy, lisää arvoparin avain-arvo rakenteeseen
- nytkään iteraattorit eivät vanhene lisäyksessä ja poistossa

Hajautustaulu, Unordered set/multiset, joka sisältää joukon alkioita ja unordered map/multimap, joka sisältää joukon alkioita, jotka assosioidaan avainarvojoukolle.

- unordered map/set muistuttavat rajapinnaltaan mapia ja setia
- tärkeimmät erot:
  - alkiot eivät ole järjestyksessä (unordered)
  - lisäys, poisto ja etsintä ovat keskimäärin vakioaikaisia ja pahimmassa tapauksessa lineaarisia
  - tarjoavat hajautuksen kannalta olennaisia funktioita, kuten `rehash(koko)`, `load_factor()`, `hash_function()` ja `bucket_size()`.

- hajautustalun kokoa kasvatetaan automaattisesti, jotta lokeroiden keskimääräinen täyttöaste saadaan pidettyä sovitun rajan alapuolella
  - hajautustaulun koon muuttaminen *(rehashing)* on keskimäärin lineaarinen pahimmillaan neliöllinen operaatio
  - koon muuttaminen mitätöi kaikki iteraattorit, muttei osoittimia eikä viitteitä

Lisäksi Standardikirjastosta löytyy joitakin muita säiliöitä:

Bittivektori `bitset<bittien_määrä>`

- `#include<bitset>`
- tarkoitettu kiinteän kokoisten binääristen bittisarjojen käsittelyyn
- tarjoaa tyypillisiä binäärisiä operaatioita (AND, OR, XOR, NOT)

Merkkijonot `string`

- `#include<string>`
- vaikka C++:n merkkijonot on optimoitu muuhun tarkoitukseen eikä niitä yleensä ajatella säiliöinä, ne ovat muun lisäksi säiliöitäkin
- säilövät merkkejä, mutta saadaan säilömään muutakin
- niillä on mm. iteraattorit, `[...]`, `.at(...)`, `.size()`, `.capacity()` ja `swap`
- merkkijonot voivat kasvaa hyvin suuriksi ja varaavat tarvittaessa automaattisesti lisää muistia

- merkkijonojen muokkausoperaatioiden (katenointi, poisto) kanssa kannattaa olla varovainen, koska niissä suoritetaan muistinvarausta ja kopiointia, minkä vuoksi ne ovat pitkille merkkijonoille varsin raskaita
- usein on muutenkin järkevää sijoittaa merkkijonot esimerkiksi osoittimen päähän sijoitettaessa niitä säiliöihin, jottei niitä turhaan kopioitaisi
- samasta syystä merkkijonot tulee välittää viiteparametreina

Säiliöiden lisäksi STL tarjoaa joukon säiliösovittimia, jotka eivät itsessään ole säiliöitä, mutta joiden avulla säiliön rajapinnan saa “sovitettua toiseen muottiin”:

Pino `stack<alkion_tyyppi, säiliön_tyyppi>`

- tarjoaa normaalien luokka-operaatioiden lisäksi vain
  - pino-operaatiot, `.push(...)`, `.top()`, `.pop()`
  - koon kyselyt `.size()` ja `.empty()`
  - vertailut "==", "<" jne.
- `.pop()` ei palauta mitään, ylimmän alkion saa tarkasteltavaksi `.top()`:illa
- pinon ylintä alkiota voi muuttaa paikallaan:
  ```
  pino.top() = 35;
  ```
- kurssin kannalta kiinnostavaa on, että käyttäjä voi valita taulukkoon tai listaan perustuvan toteutuksen
  - mikä tahansa säiliö, joka tarjoaa `back()`, `push_back()` ja `pop_back()` käy, erityisesti `vector`, `list` ja `deque`.
  - `stack<tyyppi> perus_pino;` (`deque`)
  - `stack<tyyppi, list<tyyppi> > lista_pino;`

Jono `queue<alkion_tyyppi, säiliön_tyyppi>`

- jono-operaatiot `.push(...)`, `.pop()`, `.front()`, `.back()`(!)
- mikä tahansa säiliö, joka tarjoaa `front()`, `back()`, `push_back()` ja `pop_front` käy
- muuten kutakuinkin samanlainen kuin pino

Prioriteettijono `priority_queue<alkion_tyyppi,säiliön_tyyppi>`

- lähes täysin samanlainen rajapinta kuin jonolla
- toteutus kekona
- mikä tahansa säiliö, jolla `front()`, `push_back()` ja `pop_back()` ja hajasaanti-iteraattoreita tukeva käy, erityisesti `vector` (oletus) ja `deque`
- alkioilla eri järjestys: `.top()` palauttaa suurimman
- yhtäsuurista palauttaa minkä vain
- ylintä alkiota ei voi muuttaa `top`:in avulla paikallaan
- kuten assosiatiivisilla säiliöillä, järjestämisperiaatteen voi antaa <>-parametrina tai rakentajan parametrina (strict weak ordering)

tieto-rakenne	lisäys loppuun	lisäys muualle	1. poisto	alkion poisto	n:s alkio (indeks.)	tietyn etsintä	suurimman poisto
array					$O(1)$	$O(n)_{[2]}$	
vector	$O(1)$	$O(n)$	$O(n)$	$O(n)_{[1]}$	$O(1)$	$O(n)_{[2]}$	$O(n)_{[3]}$
list	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)_{[3]}$
deque	$O(1)$	$O(n)_{[4]}$	$O(1)$	$O(n)_{[1]}$	$O(1)$	$O(n)_{[2]}$	$O(n)_{[3]}$
stack$_{[9]}$	$O(1)$			$O(1)_{[5]}$			
queue$_{[9]}$		$O(1)_{[6]}$		$O(1)_{[7]}$			
priority queue$_{[9]}$		$O(\log n)$ $_{[10]}$					$O(\log n)_{[8]}$
set (multiset)		$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(\log n)$
map (multimap)		$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(\log n)$
unordered_ (multi)set		$O(n)$ $\approx \Theta(1)$		$O(n)$ $\approx \Theta(1)$		$O(n)$ $\approx \Theta(1)$	$O(n)$ $O(n)$
unordered_ (multi)map		$O(n)$ $\approx \Theta(1)$		$O(n)$ $\approx \Theta(1)$		$O(n)$ $\approx \Theta(1)$	$O(n)$ $O(n)$

[1] vakioaikainen viimeiselle alkiolle, muuten lineaarinen
[2] logaritminen jos tietorakenne on järjestetty, muuten lineaarinen
[3] vakioaikainen jos tietorakenne on järjestetty, muuten lineaarinen
[4] vakioaikainen ensimmäiselle alkiolle, muuten lineaarinen
[5] mahdollinen vain viimeiselle alkiolle
[6] vain alkuun lisääminen on mahdollista
[7] vain lopusta poistaminen on mahdollista
[8] kysyminen vakioajassa, poistaminen logaritmisessa ajassa
[9] säiliösovitin
[10] lisäys tapahtuu automaattisesti kekojärjestyksen mukaiselle paikalle